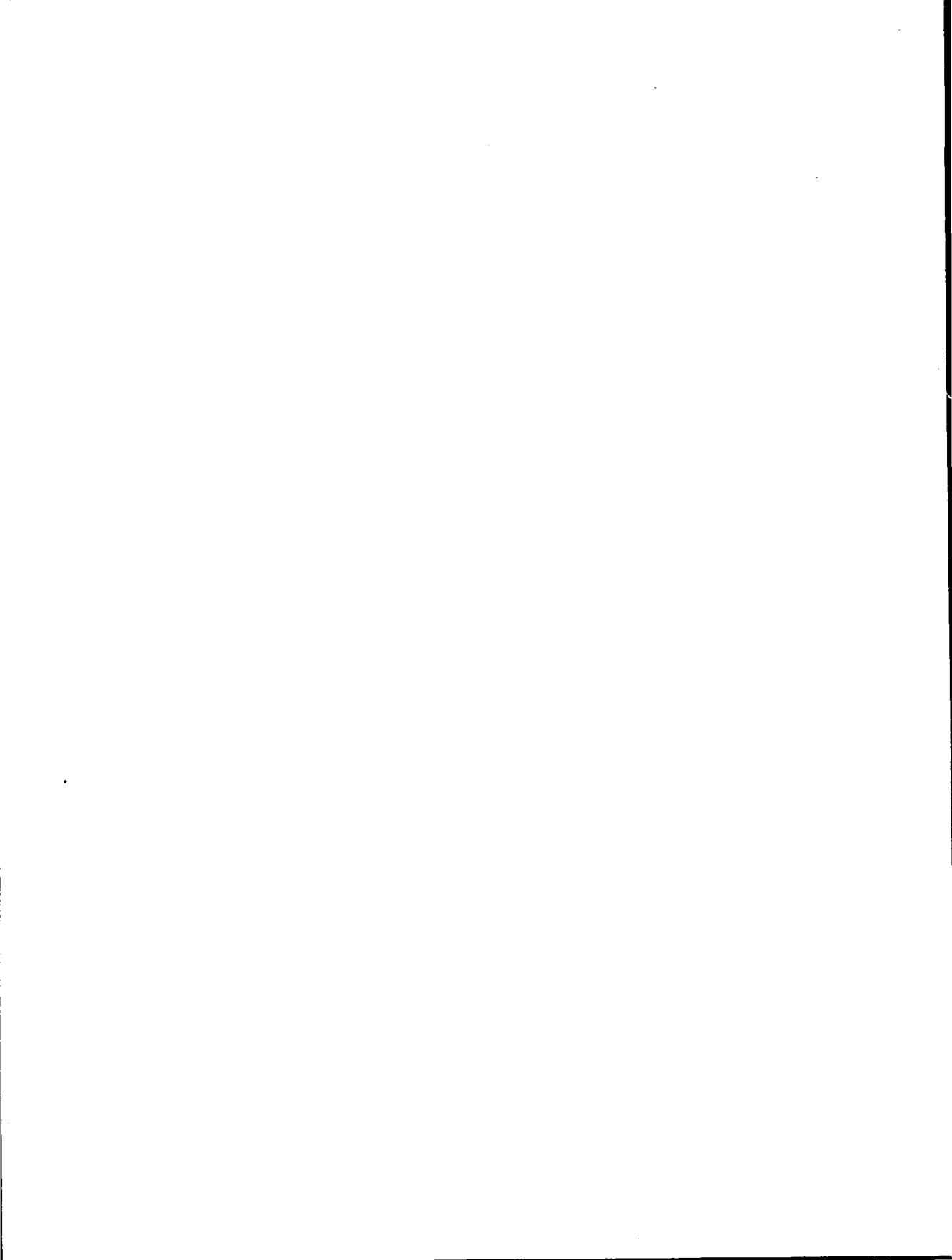


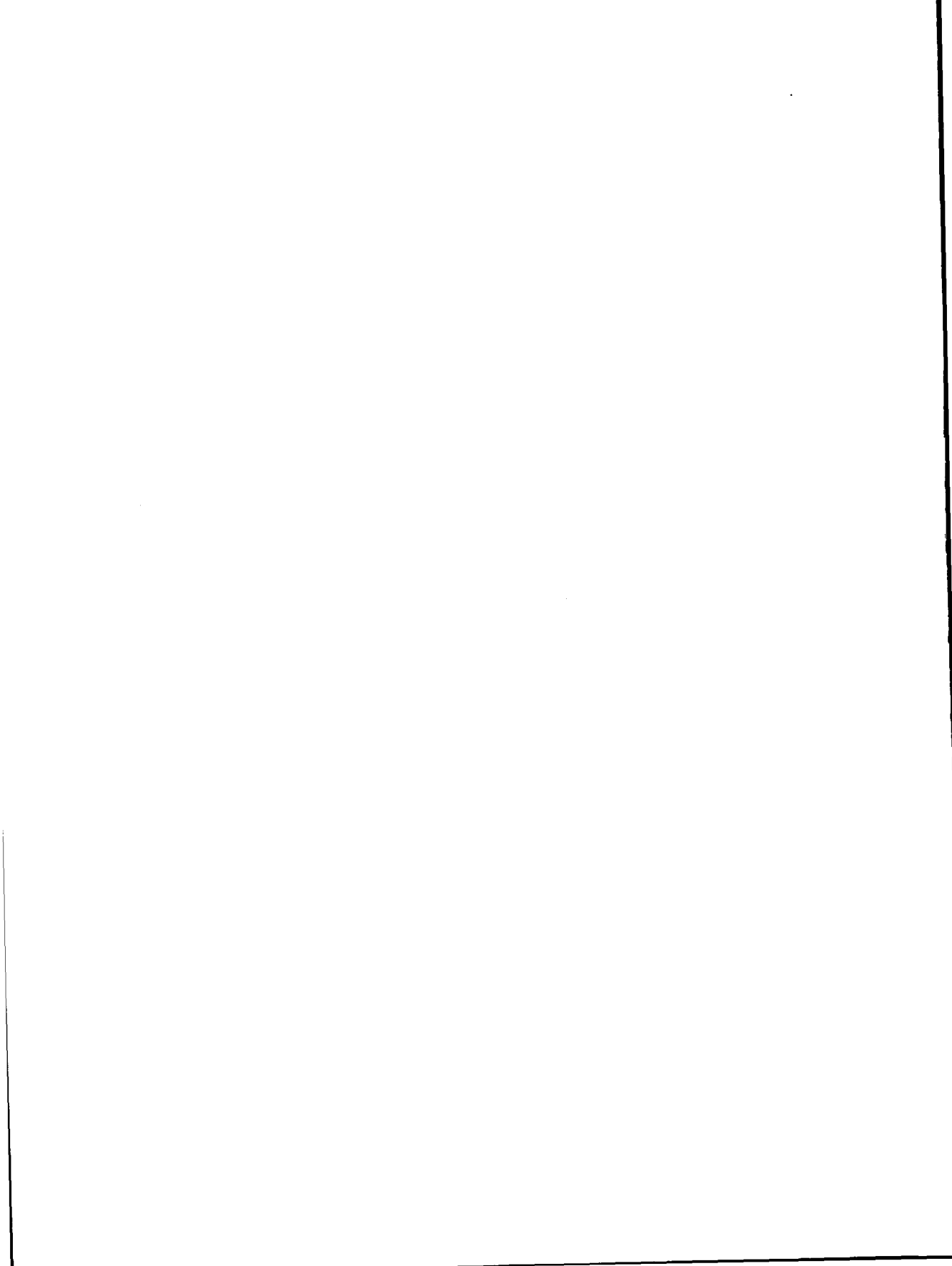
CONVEX

Exemplar Programming Guide

Second Edition



**Convex Computer Corporation**  
3000 Waterview Parkway  
P.O. Box 833851  
Richardson, TX 75083-3851  
United States of America  
(214) 497-4000



---

# Convex Exemplar Programming Guide

---

Order No. DSW-067

Second Edition

June 1995

Convex Press  
Richardson, Texas  
United States of America

---

# Convex Exemplar Programming Guide

Order No. DSW-067

Copyright © 1995 Convex Computer Corporation  
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from Convex Computer Corporation.

Although the material contained herein has been carefully reviewed, Convex Computer Corporation does not warrant it to be free of errors or omissions. Convex reserves the right to make corrections, updates, revisions or changes to the information contained herein. Convex does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

Convex and the Convex logo ("C") are registered trademarks of Convex Computer Corporation.

UNIX is a registered trademark of Novell, Inc.

Other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies, and are hereby acknowledged.



This entire book is recyclable.

Printed in the United States of America

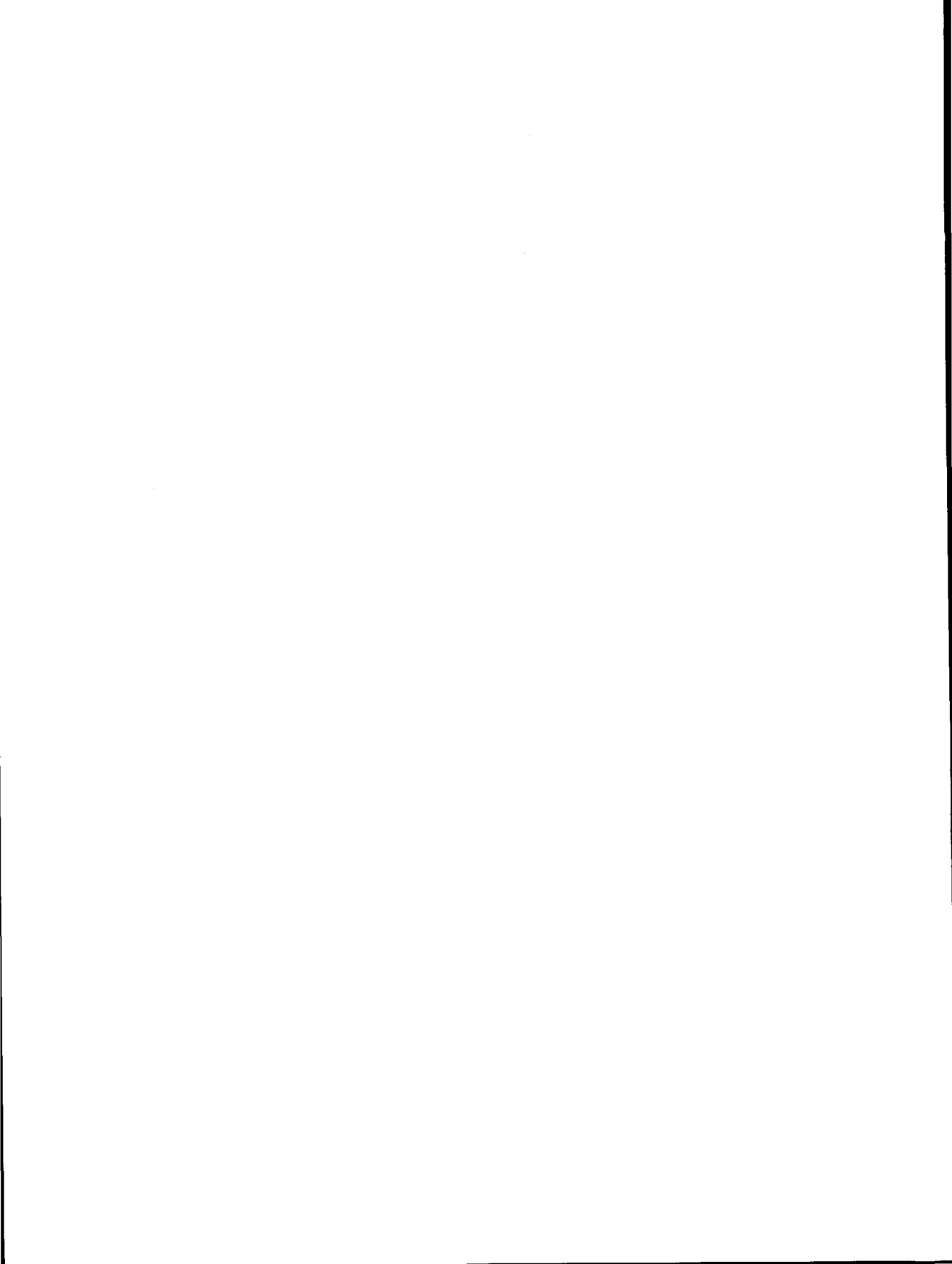
## Revision Information for

# Convex Exemplar Programming Guide

---

Edition	Document No.	Description
Second	710-030230-003	Updated to include hypernode parallelism, SPP1200 enhancements, bug fixes. Released with Convex Fortran Version 9.3 and Convex C Version 6.3.
First	710-030230-002	First production version of the Exemplar Programming Guide, released with Convex Fortran Version 9.1 and Convex C Version 6.1.

---



---

# Contents

---

<b>How to use this guide</b> .....	<b>xix</b>
Purpose and audience .....	xix
Scope .....	xix
Organization .....	xix
Suggested reading order .....	xxi
Notational conventions .....	xxii
General conventions .....	xxii
Command syntax .....	xxiii
Associated documents .....	xxiii
Ordering documentation .....	xxiv
Technical assistance .....	xxiv
Acknowledgments .....	xxiv

---

<b>1 Introduction</b> .....	<b>1</b>
Scalable parallel processing .....	1
SPP vs. traditional vector/parallel architectures .....	2
Architectural differences .....	2
Memory .....	3
Optimizing compilers .....	3
SPP vs. clustered workstations .....	3
Memory .....	4
Optimizing compilers .....	4
Interprocess communication .....	4
Peripherals .....	5
Configurability .....	5
Exemplar programming model .....	6
The shared memory paradigm .....	6
The message passing paradigm .....	7
Message passing/shared memory hybrids .....	7
Overview of Exemplar optimizations .....	8
Basic scalar optimizations .....	8
Advanced scalar optimizations .....	8
Parallelization .....	9

---

<b>2 Architecture overview . . . . .</b>	<b>11</b>
System organization . . . . .	11
Memory . . . . .	14
Physical memory . . . . .	14
Virtual memory . . . . .	14
Data caches . . . . .	16
Cache lines . . . . .	16
SPP1000 Series cache . . . . .	17
SPP1200 Series cache . . . . .	17
Cache thrashing—SPP1000 Series . . . . .	20
Cache thrashing—SPP1200 Series . . . . .	23
Cache use analysis . . . . .	25
Data alignment . . . . .	25
Interleaving . . . . .	26
Interleaving example . . . . .	28
Subcomplexes . . . . .	31
Physical configuration . . . . .	31
Subcomplex memory . . . . .	33

---

<b>3 Compiler optimizations . . . . .</b>	<b>35</b>
Optimization options . . . . .	35
-no level optimizations . . . . .	36
Instruction scheduling . . . . .	36
Span-dependent instructions . . . . .	36
Register allocation . . . . .	36
Tree-height reduction . . . . .	37
Short-circuit evaluation of conditionals in Fortran . . . . .	38
Data alignment . . . . .	39
-00 Level optimizations . . . . .	40
Instruction scheduling . . . . .	40
Redundant-assignment elimination . . . . .	40
Assignment substitution . . . . .	41
Common-subexpression elimination . . . . .	42
Redundant-use elimination . . . . .	42
Constant propagation and folding . . . . .	42
Algebraic and trigonometric simplification . . . . .	44
-01 Level optimizations . . . . .	45
Constant propagation and folding . . . . .	45
Redundant-assignment elimination . . . . .	46
Original code . . . . .	46
Optimized code . . . . .	47
Dead-code elimination . . . . .	49
Copy propagation . . . . .	49
Common subexpression elimination . . . . .	50
Code motion . . . . .	52

---

Strength reduction .....	53
Explicit arithmetic reductions .....	53
Induction variables and constants .....	54
Global register allocation .....	56
-O2 Level optimizations .....	59
Why localize? .....	60
Strip mining .....	62
Loop distribution .....	63
Loop interchange .....	64
Loop blocking .....	65
Data reuse .....	65
Reuse example .....	66
Blocking example—simple loop .....	67
Blocking example—matrix multiply .....	70
Blocking directives, pragmas and options .....	71
Loop fusion .....	72
Loop unrolling .....	75
Loop unroll and jam .....	77
IF-DO and if-for optimizations .....	81
Redundant-test elimination .....	81
Loop boundary-value peeling .....	82
Test promotion .....	84
Scalar replacement .....	85
Inhibitors of localization .....	86
Loop-carried dependencies .....	87
Aliasing .....	91
Multiple loop entries or exits .....	93
RETURN or STOP statements in Fortran .....	94
Computed or assigned GOTO statements in Fortran .....	94
Procedure calls .....	95
I/O statements .....	95
Preventing data localization .....	96
-O3 Level optimizations .....	96
Basic operation .....	96
Idle thread states .....	100
Fortran 90 constructs .....	101
Parallel optimizations .....	102
Dynamic selection .....	102
Inhibitors of parallelization .....	104
Loop-carried dependencies .....	104
Reductions .....	106
Preventing parallelization .....	107
Other parallelization directives and pragmas .....	108

---

<b>4 Basic shared memory programming . . . . .</b>	<b>111</b>
Simple manual loop and task parallelization . . . . .	111
Loop parallelization . . . . .	112
prefer_parallel . . . . .	117
loop_parallel . . . . .	118
Task parallelization . . . . .	120
Examples . . . . .	122
Critical sections . . . . .	125
-noautopar compiler option . . . . .	127
Reentrant compilation . . . . .	127
Default stack size . . . . .	128
Loop- and task- specific data privatization . . . . .	129
loop_private . . . . .	129
Using loop_private with loop_parallel . . . . .	131
Denoting induction variables in parallel loops . . . . .	132
Privatizing induction variables	
in nested loops . . . . .	134
task_private . . . . .	136
save_last . . . . .	138
Performance analysis . . . . .	140

---

<b>5 Memory classes . . . . .</b>	<b>143</b>
Private versus share memory . . . . .	144
Memory class addressing . . . . .	144
thread_private . . . . .	146
node_private . . . . .	146
near_shared . . . . .	147
far_shared . . . . .	147
block_shared . . . . .	148
Memory class assignments . . . . .	148
Static assignments . . . . .	150
thread_private . . . . .	150
node_private . . . . .	151
near_shared . . . . .	153
far_shared . . . . .	154
block_shared . . . . .	155
Dynamic assignments . . . . .	155
Memory class pointers . . . . .	155
Default classes for dynamic memory . . . . .	157
thread_private . . . . .	158
node_private . . . . .	159
near_shared . . . . .	162
far_shared . . . . .	168
block_shared . . . . .	169

---

---

## 6 Advanced shared memory programming . . . . . 177

Parallel information functions . . . . .	177
Number of processors . . . . .	178
Number of threads . . . . .	178
Number of hypernodes . . . . .	178
Number of threads on current hypernode . . . . .	179
Thread ID . . . . .	179
Hypernode ID . . . . .	180
Level of parallelism . . . . .	180
Stack memory type . . . . .	181
Thread IDs and nested parallelism . . . . .	182
Thread ID assignments . . . . .	182
Synchronization tools . . . . .	184
Gates and barriers . . . . .	184
Synchronization functions . . . . .	185
Allocation functions . . . . .	185
Deallocation functions . . . . .	186
Locking functions . . . . .	186
Unlocking functions . . . . .	187
Wait functions . . . . .	187
sync_routine directive and pragma . . . . .	188
loop_parallel(ordered) . . . . .	190
Critical and ordered sections . . . . .	192
Synchronizing code . . . . .	193
Critical sections . . . . .	193
Ordered sections . . . . .	197
Limitations . . . . .	199
Manual synchronization . . . . .	204
Advanced shared memory example . . . . .	209

---

## 7 Message passing programming . . . 215

Basic operation . . . . .	215
Message passing library . . . . .	216
Exemplar-specific configuration features . . . . .	216
Convex-specific hostfile syntax . . . . .	216
System filenames . . . . .	216
Performance tuning functions . . . . .	216
New architecture specifier . . . . .	217
PVMFRAGSIZE environment variable . . . . .	217
Other features . . . . .	217
Exemplar intertask communication . . . . .	217
ConvexPVM buffering on Exemplar . . . . .	217

Message passing approaches to parallelism . . . . .	218
Master/slave example . . . . .	219
SPMD example . . . . .	223

---

## 8 Limits of optimization . . . . . 229

Aliasing in Fortran . . . . .	230
Aliasing in C . . . . .	231
Worst-case algorithm . . . . .	231
ANSI algorithm . . . . .	232
Specifying aliasing modes . . . . .	233
Iteration and stop values . . . . .	234
Using potential aliases as	
addresses of variables . . . . .	234
Using hidden aliases as pointers . . . . .	234
Using a pointer as a loop counter . . . . .	235
Aliasing stop variables . . . . .	235
Global variables . . . . .	236
Invalid subscripts . . . . .	237
Floating-point imprecision . . . . .	237
Disabling underflow traps . . . . .	239
Misused directives, pragmas, and options . . . . .	240
Misused memory classes . . . . .	242
Improper dynamic allocations . . . . .	242
Incorrect array pointers . . . . .	245
Hidden dependencies . . . . .	246
Compiler limitations . . . . .	248
Reductions . . . . .	248
Evaluation order . . . . .	250
Incrementing by zero . . . . .	251
Nondeterminism of parallel execution . . . . .	252
Large trip counts at -O1 and above . . . . .	253
Hidden ordered sections . . . . .	253

---

## 9 Potentially unsafe optimizations. . . . 257

Simple strength reduction . . . . .	257
Code motion . . . . .	257
Conversion elimination . . . . .	258

---

## Appendix A: Compiler directives and pragmas . . . . . 261

align_cti( <i>namelist</i> ) . . . . .	262
barrier( <i>namelist</i> ) . . . . .	262
begin_tasks[ ( <i>attribute_list</i> ) ] . . . . .	262
block_loop[ ( <i>block_factor</i> = <i>n</i> ) ] . . . . .	263

block_shared( <i>allocatable_array_namelist</i> )	263
critical_section[ ( <i>gate_var</i> ) ]	263
dynsel[ ( <i>trip_count=n</i> ) ]	263
end_critical_section	264
end_ordered_section	264
end_tasks	264
far_shared( <i>namelist</i> )	264
far_shared_pointer( <i>alloc_var_name</i> )	265
gate( <i>namelist</i> )	265
loop_parallel[ ( <i>attribute_list</i> ) ]	265
loop_private( <i>namelist</i> )	266
near_shared( <i>namelist</i> )	266
near_shared_pointer( <i>alloc_var_name</i> )	267
next_task	267
no_block_loop	267
no_distribute	267
no_dynsel	268
no_fuse	268
no_loop_dependence( <i>namelist</i> )	268
no_parallel	268
no_peel	268
no_promote_test	269
no_side_effects( <i>funclist</i> )	269
no_unroll	269
no_unroll_and_jam	269
node_private( <i>namelist</i> )	269
node_private_pointer( <i>alloc_var_name</i> )	269
opt_level( <i>level</i> )	270
ordered_section( <i>gate_var</i> )	270
peel_all	270
peel	271
prefer_fuse	271
prefer_parallel	271
promote_test_all	271
promote_test	271
row_wise( <i>array_namelist</i> )	272
save_last	272
scalar	272
sync_routine( <i>funclist</i> )	272
task_private( <i>namelist</i> )	272
thread_private( <i>namelist</i> )	273
thread_private_pointer( <i>alloc_var_name</i> )	273
unroll[ ( <i>unroll_factor=n</i> ) ]	273
unroll_and_jam[ ( <i>unroll_factor=n</i> ) ]	273

---

## **Appendix B: Optimization options . . . 275**

Optimization level and related options . . . . .	275
Cross compilation options . . . . .	276
Loop replication options . . . . .	277
Loop blocking options . . . . .	278
IF-DO and if-for optimization options . . . . .	278
Register use options . . . . .	279
Data alignment options . . . . .	280
C aliasing options . . . . .	280
Other optimization options . . . . .	283

---

## **Appendix C: Optimization report . . . . 285**

Loop table . . . . .	285
Supplemental tables . . . . .	288
Analysis table . . . . .	289
Test table . . . . .	289
Privatization table . . . . .	290
Variable name footnote table . . . . .	290
Array table . . . . .	291
Examples . . . . .	291
Example 1 . . . . .	291
Loop table . . . . .	293
Analysis table . . . . .	294
Privatization table . . . . .	294
Array reference table . . . . .	294
Example 2 . . . . .	295
Loop table . . . . .	297
Analysis table . . . . .	298
Test table . . . . .	298
Privatization table . . . . .	298
Array table . . . . .	298

---

## **Appendix D: Compiler Parallel Support library . . . . . 301**

Introduction . . . . .	301
Symmetric parallelism . . . . .	301
Asymmetric Parallelism . . . . .	303
Accessing CPSlib . . . . .	305
CPS library functions . . . . .	305
Thread management functions . . . . .	305
Symmetric thread functions . . . . .	305
Asymmetric thread functions . . . . .	308
Thread information and attribute functions . . . . .	311

High-level synchronization functions .....	317
Low-level synchronization functions .....	319
sync_routine directive and pragma .....	325
Examples .....	327
Symmetric parallelism .....	327
Block parallelism .....	327
Cyclic parallelism .....	329
Asymmetric parallelism .....	330
Synchronization using high-level functions .....	332
Barriers .....	332
Mutexes .....	334
Synchronization using low-level functions .....	336
Critical sections .....	336
Ordered sections .....	337

---

<b>Glossary .....</b>	<b>341</b>
-----------------------	------------

---

<b>Index .....</b>	<b>375</b>
--------------------	------------



---

# Figures

Figure 1	Exemplar system overview	12
Figure 2	SPP Series functional block	13
Figure 3	SPP1200 Series cache architecture	18
Figure 4	Array layouts—cache-thrashing	21
Figure 5	Array layouts, non-thrashing	23
Figure 6	SPP Series memory interleaving	27
Figure 7	Interleaving of arrays A, B, and C	29
Figure 8	Hypothetical subcomplex configurations	32
Figure 9	Unbalanced tree representation	37
Figure 10	Balanced tree representation	38
Figure 11	Array storage in memory	61
Figure 12	Blocked array access	68
Figure 13	Spatial reuse of A and B	69
Figure 14	LCDs in original and interchanged loops	89
Figure 15	Values read into array A	95
Figure 16	Thread activity: one-dimensional parallelism	98
Figure 17	Conceptual strip mine for parallelization	99
Figure 18	Parallelized loop	100
Figure 19	Stride-parallelized loop	115
Figure 20	Virtual addresses for various memory classes	144
Figure 21	Physical addresses for various memory classes	145
Figure 22	Ordered parallelization	191
Figure 23	LOOP_PARALLEL (ORDERED) synchronization	199
Figure 24	Optimization report for Example 1	292
Figure 24	Optimization report for Example 1 (continued)	293
Figure 25	Optimization report for Example 2	296
Figure 25	Optimization report for Example 2 (continued)	297
Figure 26	Symmetric parallelism	302
Figure 27	Asymmetric parallelism	304



---

# Tables

Table 1	SPP Series C and Fortran optimization options . . . . .	35
Table 2	Computation sequence of A(I, J) — original loop . . . . .	87
Table 3	Computation sequence of A(I, J) — interchanged loop . . . . .	88
Table 4	Pointer class/data class combinations . . . . .	157
Table 5	Levels of parallelism . . . . .	181
Table 6	Stack type return values . . . . .	182
Table 7	Optimization report contents . . . . .	276
Table 8	Optimization report contents . . . . .	285
Table 9	Reordering transformations reported in opt. report . . . . .	287
Table 10	Optimizing/special transformations in opt. report . . . . .	288
Table 11	params->node/PARAMS(1) values . . . . .	306
Table 12	params->threadscope/PARAMS(4) values . . . . .	307
Table 13	errno values for cps_ppcall and cps_ppcalln . . . . .	308
Table 14	errno values for cps_thread_create[n] . . . . .	309
Table 15	cps_plevel return values . . . . .	312
Table 16	Accepted CMD/cmd values . . . . .	316
Table 17	WAIT(2)/wait->wait_attr values . . . . .	316



---

# How to use this guide

---

## Purpose and audience

This guide describes efficient methods for programming in Convex C and Fortran on Exemplar (also known as SPP Series) computers. The first 4 chapters cover basic concepts, including automatic optimizations and simple manual optimizations that require minimal programmer intervention. After this, more advanced topics are covered, including advanced manual optimizations, using ConvexPVM on Exemplar, and the Compiler Parallel Support Library.

The *Exemplar Programming Guide* is for experienced Fortran and C programmers. Readers need not be familiar with the Convex scalable parallel architecture, programming model, or optimization concepts; this book addresses these topics in the necessary detail.

---

## Scope

This guide covers programming methods for Convex Fortran Version 9.3 and Convex C Version 6.3, which run under SPP-UX Version 3.1 or higher. These releases of Convex Fortran and C also require the Convex SPP Series Assembler, Loader and Libraries (ALL) Version 3.6 or higher.

This guide is concerned with producing programs that efficiently exploit the features of both the Exemplar architecture and the SPP Series compilers that run on it. Producing an efficient program requires efficient algorithms and implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. This guide assumes you have chosen the best possible algorithm for your problem and helps you obtain the best possible performance from that algorithm.

---

## Organization

This document consists of the following chapters:

- Chapter 1, "Introduction," introduces the Convex Exemplar computer, discusses how it differs from other parallel

computers, presents its programming model, and provides an overview of the optimizations available using Exemplar compilers.

- Chapter 2, "Architecture overview," presents a programmer's overview of the Exemplar architecture.
- Chapter 3, "Compiler optimizations," discusses optimization levels, the options used to specify them, the automatic optimizations performed at each level, and the directives used to control these optimizations.
- Chapter 4, "Basic shared memory programming," discusses basic manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 5, "Memory classes," discusses the memory classes available for partitioning data on SPP Series machines, and how to use them.
- Chapter 6, "Advanced shared memory programming," discusses advanced manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 7, "Message passing programming," discusses message passing programming using ConvexPVM on Exemplar machines.
- Chapter 8, "Limits of optimization," discusses common optimization problems that you may encounter and provides suggestions for how to deal with them.
- Chapter 9, "Potentially unsafe optimizations," discusses the aggressive optimizations performed when the `-uo` option is specified.
- Appendix A, "Compiler directives and pragmas," lists and describes compiler directives and pragmas available for use with the SPP Series Fortran and C compilers.
- Appendix B, "Optimization options," lists and describes the optimization options available for use on SPP Series machines.
- Appendix C, "Optimization report," explains the optimization report.
- Appendix D, "Compiler Parallel Support library," discusses the Compiler Parallel Support library.
- The Glossary provides definitions of SPP-related terms.
- An Index is included at the end of the document.

---

## Suggested reading order

This book takes a bottom-up approach to presenting information. Chapters 1, 2 and 3 provide background information which will help you understand how the Exemplar architecture and compilers optimize your code; Chapter 4 tells you how to derive performance gains with minimal intervention; chapters 5 and 6 explain how to use more advanced, more complicated programming techniques to even further improve performance; Chapter 7 discusses message passing on Exemplar machines, which requires even more manual intervention. Chapters 8 and 9 tell you about problems you may encounter when using the techniques of the previous chapters and how to enable even more aggressive optimizations. The appendixes contain mostly reference information, including a discussion of the Compiler Parallel Support library (CPSlib), which is the most programmer-dependent optimization tool available for SPP Series computers.

If you are interested in a general, comprehensive overview of programming for the Exemplar computer, read the chapters in order.

If you are interested in simply compiling existing programs and getting them to run with minimal effort, start with chapters 3 and 8; following the cross references that interest you will probably expose you to as much of the rest of the book as is necessary.

If you are interested in getting maximum performance gains for minimum programming effort, start with chapters 3 and 4, and proceed if necessary.

If you are willing to spend some time adding directives and rewriting some of your code to realize significant performance benefits (especially if your Exemplar system is equipped with multiple hypernodes), read at least chapters 2 through 6.

If you are interested in running message passing codes on your Exemplar system, start with Chapter 7; you may also want to read chapters 2 through 6 to see how the compilers can help you with automatic optimizations.

If you are interested in very low-level control over parallelism using the Compiler Parallel Support library, start with Appendix D. Again, you may want to refer to the other chapters to see how the compiler can help with automatic optimizations.

---

## Notational conventions

This section discusses notational conventions used in this book.

---

### General conventions

In general, the following conventions are used in this guide:

- *Italic*
  - Designates user-supplied variables in a command-line or code example
  - Introduces new and important terms
  - Identifies variables in mathematical equations
  - Indicates document titles
- Constant-width font designates input and output, including
  - Command names and options
  - System calls
  - Data structures and types
  - Variables and arrays
  - Function and subroutine names
  - Directives, program statements, display examples, printout examples, and error messages returned

Note that except where noted, the directives and pragmas described in this book can be used with both the C and Fortran compilers. In general discussion, these directives and pragmas are presented in lower case type, but either compiler will recognize them regardless of their case.

- **Bold constant-width font** designates text that must be input by the user.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.

References to man pages appear in the form `mnpname(1)`, where “mnpname” is the name of the man page and is followed by its section number enclosed in parentheses. To view this man page, you would type:

```
man 1 mnpname
```

## Note

A Note highlights important supplemental information.

---

## Caution

---

A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

---

### Command syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
```

1. COMMAND must be typed as it appears.
2. *input\_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional, optional input file names may be supplied.
4. Either a or b must be supplied.
5. [*output\_file*] indicates an optional file name.

---

### Associated documents

Convex Computer Corporation provides the following documents to help you use the Fortran and C compilers and associated tools:

- For more information about the Convex Fortran compiler, refer to the *Fortran User's Guide* (DSW-038), *Fortran Language Reference Manual* (DSW-037), and *Release Notice, Fortran Compiler V9.1*.
- For more information about the Convex C compiler, refer to the *C User's Guide* (DSW-086) and *Release Notice, Convex C V6.1*.
- For more information about CXpa, refer to the *CXpa Reference for Exemplar Systems* (DSW-605).
- For more information on the CXdb debugger, refer to the *CXdb Reference* (DSW-472).
- For more information on the Exemplar architecture, refer to the *Exemplar SPP1000/1200 Architecture* manual (DHW-014).
- For more information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (DSW-853).
- For more information on using SPP-UX, refer to the *HP-UX Reference* (Hewlett-Packard order number B2355-90004).

---

## Ordering documentation

To order this document or any other Convex document, send requests to:

Convex Computer Corporation  
Customer Service  
P.O. Box 833851  
Richardson, Texas 75083-3851 U.S.A.

Order documents by title, requesting the most recent edition. In some situations, you may not want the current edition. To receive a specific edition of a manual, contact the local Convex office or call the Technical Assistance Center (TAC).

---

## Technical assistance

If you have questions that are not answered in this book, contact the Convex Technical Assistance Center (TAC):

- Within the continental U.S., use (800)952-0379.
- Outside continental U.S., contact your local Convex office.

---

## Acknowledgments

This book is the product of unprecedented cooperation among Convex development, marketing, support and technical writing staffs. While it is impossible to list every person that contributed ideas to the *Exemplar Programming Guide*, the author would like to especially thank the following people:

- Joel Williamson, whose *SPP-1 Programming Model* document served as a catalyst, basic outline, and technical reference for this book. Without his guidance and patience, this book could never have been written.
- Greg Astfalk, who provided the basis for Appendix A and spent many hours explaining concepts and reviewing drafts.
- Aaron Potler, who provided understandable explanations of complex features and performed thorough, timely reviews.
- The compiler, operating system and library development teams, who provided countless (often repetitive) explanations and spent many hours reviewing the book: Gary Applegate, Rich Bleikamp, Terry Caracuzzo, Robert Cox, Tony Linthicum, Randy Meyer, Geoffrey Rogers, Perry Schmidt, Mark Schroeder, and Bill Torkelson.
- The technical reviewers, whose comments improved the accuracy and readability of the book: Clare Bernier, Marianne Chamberlain, Robert Ellis, James Hansen, Paco Romero, Adam Schwartz, Sharon Shaw and Kim Silva-Lepe.
- The editors, who improved the overall quality of writing in the book: Jerry Hill and Kristi Knox.

- The managers who provided the vision, resources and guidance necessary to make this book and the compilers it documents possible: Matt Blanton, Steve Fieler, Marilyn Lutz, Tom McClendon, Steve Rowan, and Presley Smith
- The additional Convex personnel and preferred partner customers that offered suggestions and opportunities to solicit them: Frank Baetke, Don Davis, Wayne Greenberg, Phil Merkey, Daniel Savarese, and Thomas Sterling.
- Numerous other Convex personnel and customers who provided feedback, suggested changes and submitted bugs against the first edition.

—Vince Cavasin, June 1995



The distinguishing characteristic of Convex supercomputers has always been their ease-of-use. The Convex Exemplar family of computers brings this ease-of-use legacy to a massively parallel architecture. With the Exemplar family, the full power of massive parallelism can finally be realized without the need to employ complicated and highly platform-dependent programming paradigms. Exemplar compilers generate efficient parallel code with very little intervention on your part; this efficiency can be even further increased by using the techniques discussed in this book.

This chapter provides a general overview of the Exemplar architecture as compared to other parallel architectures, the optimizations available on Exemplar, and the applicable programming models.

---

## Scalable parallel processing

Convex Exemplar systems implement massively parallel processing (MPP) using scalable parallel processing (SPP) technology. The Exemplar SPP1000 Series is the first generation in the Exemplar line. It consists of the compact SPP1000/CD system, and the full-size SPP1000/XA system. The SPP1200 Series is the second generation Convex SPP system. It consists of the compact SPP1200/CD system, and the full-size SPP1200/XA system.

As the name implies, scalable parallel machines can be scaled to meet your specific needs. Many MPP systems can only be expanded by doubling the number of processors, or require a large number of processors in their base configuration; Convex Exemplar SPP Series systems can contain as few as 2 processors and as many as 128, and can be scaled by as few as 2. These processors are arranged in 1 to 16 hypernodes, with each hypernode containing 2 to 8 processors. Processors communicate with each other, with memory and with I/O devices via the Coherent Toroidal Interconnect (CTI). The CTI consists of a nonblocking crossbar on each hypernode for intrahypernode

communication, and four high-speed CTI rings which link the hypernodes together for interhypernode communication. The CTI ring design is derived from the IEEE standard 1596-1992, SCI (Scalable Coherent Interface), but this Exemplar implementation sacrifices complete SCI compatibility to provide lower latencies.

SPP Series memory is also easily scalable; each hypernode can support 128 Mbytes, 256Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. This allows for a maximum of 32 Gbytes on a 16-hypernode system. Each process can access its full 32 bit (4 Gbyte) virtual address space, and, if necessary, programs can be written in a manner that allows them to surpass the 4 Gbyte limit and access all the memory on a system.

---

## **SPP vs. traditional vector/parallel architectures**

Scalable parallel processing represents a departure from traditional vector/parallel supercomputers like the Convex C Series. The C Series architecture is used to illustrate the difference between traditional and SPP architectures below, but the same differences apply in principle to all vector/parallel machines.

### **Architectural differences**

C Series machines contain a limited number (1-8) of custom processors connected by a high-speed crossbar to a large, shared memory. For connecting small numbers of processors such as these to memory, cross bars are cost-effective and fast, allowing all processors to access all memory with equally high speed. Each processor is equipped with one or more vector units which speed loop computations involving arrays by performing array arithmetic on up to 128 elements per vector instruction. Machines containing multiple CPUs can further reduce time to solution by adding parallelism at the process, loop, and task level.

The SPP Series architecture takes a different approach. The SPP1000/CD systems support 2, 4, 8, 12 or 16 Hewlett-Packard PA-RISC 7100 microprocessors in one or two hypernodes. The SPP1000/XA system supports 4 to 128 PA-RISC 7100 processors in 1 to 16 hypernodes. The SPP1200/CD systems support 2, 4, 8, 12 or 16 Hewlett-Packard PA-RISC 7200 microprocessors in one or two hypernodes. SPP1200/XA system supports 4 to 128 PA-RISC 7200 processors in 1 to 16 hypernodes. Rather than using vector units to exploit fine-grain parallelism, these processors speed scalar processing by using a reduced set of high-speed instructions coupled with pipelining, high-speed instruction and data caches, and a large register set.

Two-dimensional parallelism, which can benefit nested parallel structures, is also possible on multihypernode SPP Series machines. Rather than implementing the first dimension in the vector unit and the second across processors (as in C Series), the SPP Series can implement the first level within a hypernode and the second across hypernodes. Single-dimensional parallelism that spans hypernodes can also be implemented.

### Memory

Because of the potentially large number of processors available on a multihypernodal SPP Series system, memory access via a system-wide crossbar is not practical. Instead, low latency, high-bandwidth memory access is provided by globally shared memory (GSM). In this model, physical memory is distributed among all hypernodes, and the entire virtual address space of a process is accessible by every processor. Processors within a hypernode can access hypernode-local memory via the crossbar regardless of whether the address space is on one or more hypernodes; memory in another hypernode can be accessed via the CTI rings. Of course, interhypernode accesses take longer than intrahypernode accesses. However, part of every hypernode's memory is dedicated to act as a CTIcache, which holds copies of commonly used data from other hypernodes. These CTIcaches and the processor caches are *coherent*, meaning that when a thread references a data item via its virtual address, the value it receives will be the most recently-assigned value. By holding frequently-referenced data close to its referencing processes, regardless of the actual memory location of the data, these caches provide excellent data distribution.

### Optimizing compilers

Programs that optimize well on traditional vector/parallel machines will optimize well on the SPP Series with little manual intervention. Convex SPP Series compilers are designed to automatically exploit opportunities for parallelism and data localization in programs written for shared memory machines. Chapters 3 through 7 discuss manual optimizations that can yield even more performance from such programs.

---

### SPP vs. clustered workstations

While the SPP Series architectures use the same processors found in HP workstations, its low-latency GSM, automatic optimizing compilers, high-speed interconnections, shared peripherals, and user-configurability sharply distinguish it from clustered

workstations. The following subsections discuss each distinguishing feature in detail.

## **Memory**

Each workstation in a cluster has its own private memory; there is no shared memory. This means that any data that must be shared among processors must be passed over the low-performance network that connects them. While Exemplar can support this method of programming, it offers the many advantages of GSM, as described in the “SPP vs. traditional vector/parallel architectures” section.

Many workstation operating systems reserve a large amount of memory for system use, restricting user processes to what’s left. SPP-UX only requires a small fraction of each processor’s memory, leaving a large majority of it for user processes, whether they are using GSM or message passing.

## **Optimizing compilers**

Programs for clustered workstations are compiled using the workstations’ compilers. If the cluster contains workstations that require different executables (i.e. if it is a *heterogeneous* cluster), the programmer must generate the executables using the proper compiler. Homogeneous clusters eliminate this requirement, but automatic parallelization is nevertheless unavailable on any type of cluster. The compilers used may generate efficient code for each processor, but any parallelism or process coordination must be explicitly implemented by the programmer via message passing.

Convex compilers have long been highly regarded for their ability to automatically optimize code. Convex Exemplar compilers continue this legacy and add to it fully automatic node- and thread-wise parallelism and several new data localization optimizations designed to improve memory usage and aid parallelization. Additionally, a rich set of directives allows you to even further enhance the automatic optimizations performed on your shared memory program.

Convex Exemplar compilers give the highest performance with little or no programmer intervention from generic programs which exploit GSM. Message passing programs, with their parallelism explicitly coded, will also benefit from all Exemplar compiler optimizations.

## **Interprocess communication**

To communicate among themselves or access each other’s data, the processors in a cluster of workstations must communicate over low-performance networks and access distributed memory.

Communication can only be handled by passing explicit messages between workstations over the network; because of the distributed memory and absence of parallelizing compilers, programmers must explicitly code parallelism. Parallel tasks running on clusters, then, must be fairly autonomous to avoid wasting time waiting for data or synchronization instructions to travel over the network. Clusters are best suited to coarse-grained parallelism, such as that possible at the process level, or to manually-parallelizable algorithms that contain a large ratio of computation to communication. In these cases, task chunks or processes and their data are parcelled out to underused workstations, run to completion, and the results are sent back to the parent.

Fine-grained, loop-level parallelism is difficult to do efficiently on clusters because of the need for frequent data accesses and synchronization.

Exemplar systems are suitable for both coarse- and fine-grained parallelism. Programs containing potential parallelism will, when compiled with Convex Exemplar compilers, automatically exploit the parallelism, spawning threads to run on as many processors (or even hypernodes) as are available, and rejoining these threads upon completion. This fine-grained parallelism will take full advantage of Exemplar's fully coherent caches and high-speed interconnects. While message passing is supported, and can be used to speed certain applications (refer to Chapter 7, "Message passing programming"), with GSM, it is not necessary for most programs. When message passing is used on Exemplar, the high-speed interconnects can give a substantial performance increase over traditional networks. This makes message passing programs that exploit finer-grained parallelism practical.

SPP-UX automatically schedules threads to idle and underused processors as necessary, ensuring a balanced machine load and exploiting both thread- and process-level parallelism.

### Peripherals

Peripheral devices connected to an Exemplar system can be accessed from any processor on the machine. On clustered workstations, peripherals are processor-dependent. Programs running on Exemplar systems therefore have access to potentially greater mass storage space.

### Configurability

Exemplar systems can be configured by system administrators into one or more *subcomplexes*. A subcomplex is a virtual machine consisting of a specified number of processors on specified

hypernodes. Subcomplexes allow Exemplar systems to be configured into several virtual machines which make the most effective use of available resources given the computational tasks at hand. Subcomplexes can be reconfigured as needed without rebooting the machine; this allows time- or task-dependent reconfigurations, or immediate correction of inefficient configurations. No similar capability is available for clusters.

In terms of configuring hardware, adding processors to a cluster can actually degrade performance because of the low-performance network and private memory. The network can present a bottleneck when parallelism increases to exploit the new processors; to overcome this, coarser granularity can be used—and this can require more private memory than the processors can address. The absolute performance of Exemplar, on the other hand, increases unhindered by a traditional network or private-memory limits. Adding peripherals and memory to an Exemplar system can also provide improved absolute performance, since all processors can access both, whereas memory and peripherals are processor-specific on clusters.

---

## **Exemplar programming model**

The Exemplar programming model provides two perspectives from which a programmer can write (or adapt) code to run on Exemplar. Those perspectives are the shared memory and message passing paradigms.

---

### **The shared memory paradigm**

In the shared memory paradigm, the compilers handle optimizations, and, if requested, parallelization. Data distribution is taken care of by hardware via the CTIcache. Numerous compiler directives and pragmas (which are discussed in detail in Chapter 4, “Basic shared memory programming,” and Chapter 5, “Memory classes,” and listed in Appendix A, “Compiler directives and pragmas”) are available to even further increase optimization opportunities. Since the shared memory approach is used on most non-MPP machines, Exemplar’s support of it allows programs from such machines to be easily ported to the Exemplar architecture. Most programmers will also use the familiar shared memory paradigm for programs written from scratch for Exemplar, since it automates many of the tasks for which the message passing paradigm requires explicit coding.

Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming,” cover shared memory programming in detail.

---

## The message passing paradigm

The Exemplar message passing paradigm uses a version of the ConvexPVM (Parallel Virtual Machine) library, which is also available for use with clustered workstations. However, on Exemplar, with its tightly-coupled processors, interprocess communication is much faster, and finer-grain parallelism is more practical.

Under the message passing paradigm, ConvexPVM functions are used to explicitly spawn parallel processes, share data among them, and coordinate their activities. There is no shared memory; each process has its own private 4Gbyte address space (this is determined by the HP 7x00's 32-bit address space) and any data that must be shared must be explicitly passed between processors. Since processes cannot access each other's memory space, cache coherency and its inherent duplication of data is not necessary, so a larger amount of physical memory can be made available to the program as a whole.

Support of message passing allows programs written under this paradigm for distributed memory machines to be easily ported to Exemplar. Programmers familiar with using the message passing style may choose to write new programs for Exemplar using this paradigm rather than shared memory, and they can realize a substantial performance boost over conventional message-passing machines, even when coding finer-grain parallelism. Lastly, the few programs that require more per-process memory than possible using shared memory will benefit from the manually-tuned message passing style.

Chapter 7, "Message passing programming," covers message passing programming in detail.

---

## Message passing/shared memory hybrids

Some programs may benefit from combining the paradigms to allow several shared memory processes to coordinate their activities via message passing. This model allows the majority of the program to be written in the familiar shared memory style while the process-private memory benefits of message passing are exploited.

---

## Overview of Exemplar optimizations

Convex Exemplar compilers perform a broad range of user-selectable optimizations. These optimizations, which are specified via compiler command line options, are briefly introduced here; a more thorough discussion, including the options associated with each, is given in Chapter 3, "Compiler optimizations."

---

### Basic scalar optimizations

Basic scalar optimizations improve performance at the basic block and program unit level.

A basic block is a sequence of statements into which there is a single entry point and out of which there is a single exit. Branches do not exist within the body of a basic block. A program unit is a subroutine, function or main program in Fortran or a function (including `main`) in C; program units are also often generically referred to as procedures. Basic blocks are contained within program units; program unit level optimizations span basic blocks.

To improve performance, basic scalar optimizations perform the following:

- Fully exploit the processor's functional units and registers
- Reduce the number of times memory is accessed
- Simplify expressions
- Eliminate redundant operations
- Replace variables with constants
- Replace slow operations with faster equivalents

---

### Advanced scalar optimizations

Advanced scalar optimizations are primarily intended to maximize processor data cache usage. This is referred to as data localization. Concentrating on loops, these optimizations strive to encache the data most frequently used by the loop and keep it encached so as to avoid costly main memory accesses.

The compilers can perform a number of transformations on loops to facilitate more efficient data localization. The most fundamental of these transformations is *blocking*. Blocking breaks a loop whose data is too large to fit entirely into the processor data cache into a loop nest; the inner loop can then make efficient use of the cache while the outer loop runs the inner loop as many times as necessary to cover the entire original trip count.

Advanced scalar optimizations include several other loop transformations; many of them either facilitate more efficient strip mining or are performed on strip mined loops to optimize processor data cache usage. All of these optimizations are covered in detail in Chapter 3, "Compiler optimizations."

Advanced scalar optimizations implicitly include all basic scalar optimizations.

---

## Parallelization

It is through parallelization that you can realize the full power of a scalable parallel computer like Exemplar. Parallelization allows a program to be executed by as many processors as are available within its subcomplex, in most cases dramatically reducing time-to-solution. Convex Exemplar compilers can automatically locate and exploit loop-level parallelism in most programs, and, using the techniques described in Chapter 5, "Memory classes," you can assist the compilers in finding even more parallelism in your programs.

Loops that have been data-localized are prime candidates for parallelization; individual iterations of inner loops that contain strips of localizable data can be parcelled out among several processors and run simultaneously. The maximum number of processors that can be used is limited by the number of iterations of the outer loop, and, of course, by processor availability.

While most parallelization is done on nested, data-localized loops, other code can also be parallelized. For example, through the use of manually-inserted compiler directives, sections of code outside of loops can also be parallelized.

Parallelization optimizations implicitly include all scalar optimizations.



This chapter provides an overview of the Exemplar system architectures for programmers. Both the SPP1000 Series and SPP1200 Series computers are covered. For more detailed information on the topics discussed in this chapter, refer to the *Exemplar SPP1000/1200 Architecture* manual, order number DHW-014.

---

## System organization

Think of an Exemplar system as a shared memory computer with two tiers of memory latency. The hypernode crossbar constitutes the first tier, and the CTI rings constitute the second. Exemplar SPP1000/XA and SPP1200/XA systems consist of 1 to 16 hypernodes. Each hypernode contains 4 or 8 processors and 256 Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. Processors are arranged in functional blocks, each containing two processors, 128 Mbytes to 512 Mbytes of memory, and some control devices. Functional blocks within a hypernode communicate with each other, with memory, and with peripherals via a 5 port nonblocking crossbar. Functional blocks communicate across hypernodes via four CTI rings.

Exemplar SPP1000/CD and SPP1200/CD systems consist of 1 or 2 hypernodes, each containing 2, 4 or 8 processors, but are otherwise identical to their respective /XA systems.

Figure 1 shows an overview of a multihypernode Exemplar system containing 8 processors per hypernode.

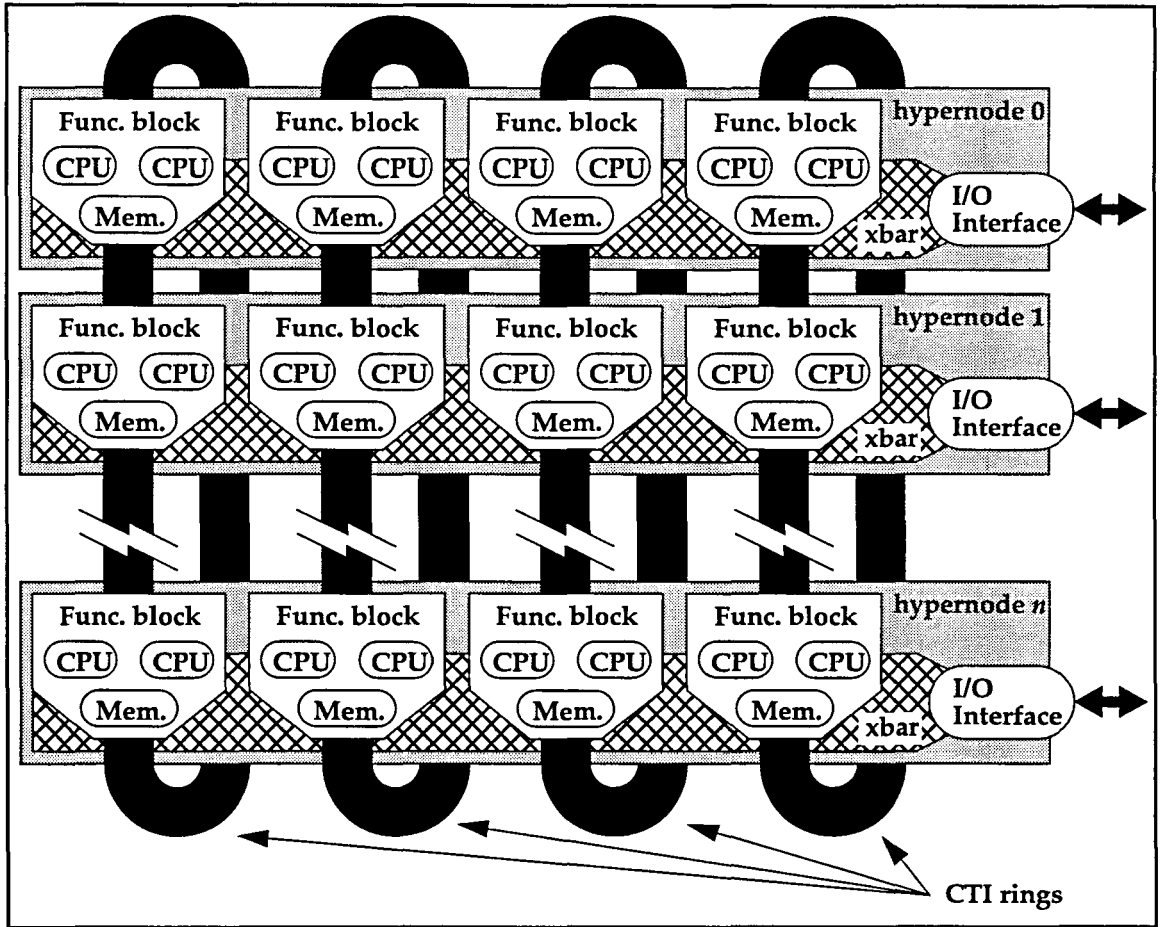


Figure 1 Exemplar system overview

Though it is difficult to illustrate in two dimensions, hypernodes are distributed evenly around the CTI rings. While the rings attach specific functional blocks within the hypernode as illustrated, any functional block can access memory on any other functional block by routing its request through its own crossbar to the functional block that is attached to the correct ring. Data is returned via the same ring and then routed via the crossbar back to the requesting functional block.

Figure 2 shows a single functional block in more detail.

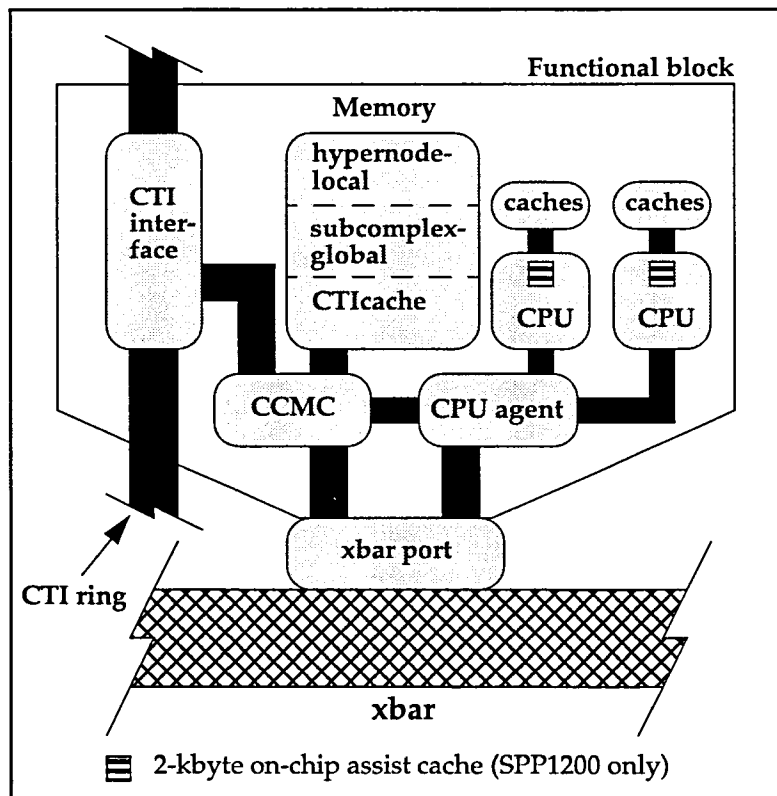


Figure 2 SPP Series functional block

CPUs communicate directly with their own instruction and data caches, which can be accessed by the CPU in 1 clock (assuming a full pipeline). SPP1000 Series machines use 1-Mbyte off-chip instruction and data caches; SPP1200 Series machines use 256-kbyte off-chip instruction and data caches, and a 2-kbyte on-chip assist data cache (both SPP1200 Series caches can be accessed in 1 clock). The functional block's two CPUs communicate with the rest of the machine through the CPU agent. The Convex Coherent Memory Controller (CCMC) provides the interface between the functional block's 2 memory banks and the rest of the machine. All intrahypernode memory accesses take around 600 nanoseconds of wall time, even if they can be fulfilled from the functional block's own memory block. This is because they must traverse the crossbar, which gives equal access to all hypernode memory from all functional blocks.

---

## Memory

Each shared memory process running on an SPP Series system accesses its own 4 Gbyte virtual address space. 3.75 Gbytes of this space are available to hold program text, data and the stack; the remaining .25 Gbyte is used by SPP-UX. The stack size is tunable; refer to the "Default stack size" section of Chapter 4, "Basic shared memory programming," for more information.

Processes cannot access each other's virtual address space. This virtual memory maps to the physical memory of the subcomplex on which the process is running. Physical memory is partitioned on a subcomplex basis; subcomplexes cannot access each other's physical memory, just as processes cannot access each other's virtual memory.

### Physical memory

All memory (excluding processor caches) on SPP Series systems is implemented in the memory banks of the functional blocks. Each functional block contains 2 memory banks, for a total of 8 banks per 8-CPU hypernode. As shown in Figure 2, this memory is typically partitioned (by the system administrator) into hypernode-local, subcomplex-global, and CTIcache. It is also interleaved as described in the "Interleaving" section later in this chapter.

Hypernode-local memory, as its name implies, is local to its hypernode, and cannot be accessed by other hypernodes. This is where application and SPP-UX executables, as well as user process data that has been explicitly declared private, reside.

Subcomplex-global memory is accessible by all CPUs in a given subcomplex. This memory can be allocated as discussed in the "Subcomplexes" section, which follows.

The CTIcache is used to store copies of global data fetched from other hypernodes.

### Virtual memory

Virtual memory is divided into 5 classes. The compilers will choose default classes for your programs that provide good performance; programmers can also manually assign data to memory classes to improve data distribution and further increase performance. Note, however, that doing so also requires some other aspects of optimization, particularly loop parallelization, to be handled manually.

Brief descriptions of the virtual memory classes and their physical memory mappings follow.

### `thread_private`

This memory is private to each thread of a process. A `thread_private` data object has a unique virtual address for each thread within its hypernode. These addresses map to unique physical addresses in hypernode-local physical memory on each hypernode. Threads access the physical copies of `thread_private` data residing on their own hypernode when they access `thread_private` virtual addresses.

### `node_private`

This memory is private to the threads running on a given hypernode. A `node_private` data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode.

### `near_shared`

Data objects of the `near_shared` class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. All data of a `near_shared` object maps to single physical addresses on a particular hypernode.

### `far_shared`

Data objects of the `far_shared` class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, `far_shared` data is distributed by pages round-robin to all the hypernodes in the subcomplex, so the virtual address maps to a single physical address located on one of the hypernodes.

### `block_shared`

Data objects of the `block_shared` class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, `block_shared` data is distributed in blocks equally among the hypernodes on which the process is executing (which could be a subset of the subcomplex), one block per hypernode. `block_shared` memory must be dynamically allocated; the programmer can then easily ensure that threads on a hypernode make most of their accesses to the block residing on their hypernode.

Using these memory classes is discussed in detail in Chapter 5, "Memory classes."

---

## Data caches

SPP1000 Series and SPP1200 Series machines each use high-speed data caches to improve performance, but the architectures differ in their implementation of the cache.

### Cache lines

Before examining the specifics of each architecture's caches, you must understand how data is moved between the cache and memory. A *cache line* describes the size of a chunk of contiguous data that must be copied into or out of a cache in one operation. When a processor experiences a cache miss—that is, requests data that is not already encached—the cache line containing the address of the requested data is moved to the cache. This cache line also contains some number of other data objects which were not specifically requested; this number varies according to the object size and the type of cache line in question.

A *CTIcache line* is 64 bytes long, so 64 bytes of data move from global shared memory to the CTIcache when a CTIcache miss occurs. A CTIcache line consists of two contiguous *processor cache lines*, which are 32 bytes long. When a processor cache miss occurs, the requested data is fetched as part of a contiguous 32 byte cache line. If this data resides in any memory on the processor's hypernode, it need not traverse the CTIcache; if it resides in the memory of another hypernode, it will be fetched through the CTIcache.

All processor-encached data not residing on the processor's hypernode must pass through the CTIcache, so if this data is contained in processor cache, it is replicated in the CTIcache.

A primary reason cache lines are employed is to allow for *data reuse*. Data in a cache line is subject to reuse if, while the line is encached, any of the data elements contained in the line besides the requested element are referenced by the program, or if the requested element is referenced more than once.

Because data can only be moved to and from memory as part of a cache line, both load and store operations cause their operands to be encached. Cache coherency hardware invalidates cache lines when they are stored to by a particular processor; this indicates to other processors that need the data they contain that they must load the cache line from the processor that most recently wrote to it.

### SPP1000 Series cache

SPP1000 Series machines use 1-Mbyte write-back direct-mapped data caches. In a direct-mapped cache, the cache address for a given data object is a function of the object's full virtual address. On the SPP1000 Series, cache addresses are computed within a process using the following formula:

$$\text{cache\_address} = \text{MOD}(\text{virtual\_address}, 2^{20})$$

Where the MOD function yields the remainder when *virtual\_address* is divided by  $2^{20}$ . The value of  $2^{20}$  is 1,048,576, or 1 Mbyte. Thus, a data object's cache address is the least-significant 20 bits of its virtual address.

This addressing scheme can result in *cache thrashing*, which is discussed in the "Cache thrashing—SPP1000 Series" section.

### SPP1200 Series cache

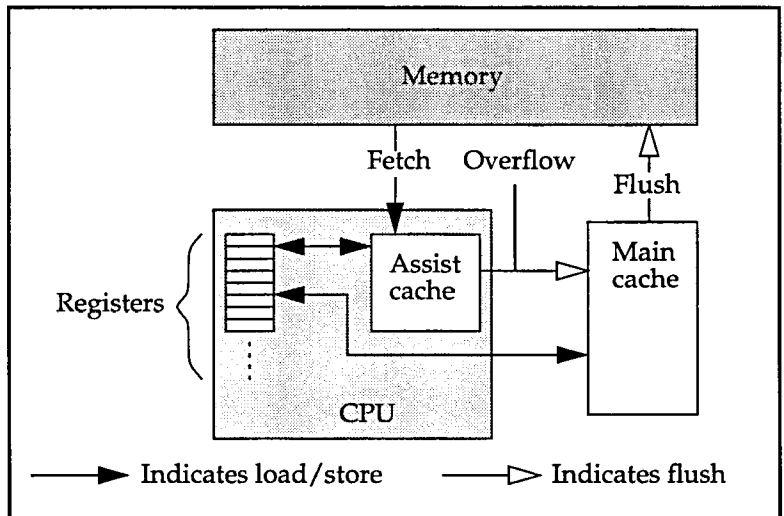
The SPP1200 Series architecture uses two data caches—a 256-kbyte off-chip main cache, and a 2-kbyte on-chip assist cache.

The 256-kbyte off-chip write-back direct-mapped data cache is, except for size, identical to the off-chip data cache used in the SPP1000 Series architecture. Cache addresses in the SPP1200 Series main cache are determined in the same manner as for the SPP1000 Series except that the least-significant 18 bits of the data's virtual address ( $2^{18} = 262144$  or 256 kbytes) are used rather than the least significant 20 bits.

The 2-kbyte on-chip assist cache is effectively fully associative. In a fully-associative cache, there is no direct correlation between a data item's address in memory and its address in the cache, so an item can never be displaced due to cache thrashing. A data item's address is determined by a hashing algorithm, and all cache addresses are used to store data. No data is displaced from the assist cache until it is full, and in that case, the algorithm will displace the data item that was fetched least recently (i.e. the algorithm implements a FIFO queue). Data displaced from the assist cache is moved to the main cache; all data fetched from memory enters the assist cache first and only enters the main cache if it is displaced from the assist cache.

The assist cache holds 64 processor cache lines ( $64 \times 32$  bytes = 2048 bytes, or 2 kbytes).

The SPP1200 Series cache architecture is illustrated in Figure 3.



**Figure 3** SPP1200 Series cache architecture

Keep in mind that both load and store operations require the cache line containing the data being loaded or stored to be encached in either the assist cache or the main cache. The term *reference*, as used in the following discussion, refers to either a load or store.

When a data item is referenced, all 64 assist cache lines plus one main cache line (the line containing the appropriate virtual address) are checked simultaneously. For a load, if the data resides in either cache, it is loaded directly from the cache to a register in 1 clock cycle (assuming a full pipeline). Data from the main cache cannot pass through the assist cache in this case; the same data item can never exist in both the assist cache and the main cache. However, for any reference, if the data does not reside in either cache, it must be fetched from main memory (or the CTIcache). In this case, data satisfying a load is always copied into the assist cache on its way to a register.

There is no path from memory directly to the main cache. All entries in the main cache pass through the assist cache when initially fetched; the main cache only receives data displaced from a full assist cache.

When a store executes, the data is written to whichever cache its cache line resides on. This does not necessarily imply that data is stored to the cache from which it was loaded; it is possible that between the load and store operations a cache line residing in the assist cache may be flushed to the main cache, or that a line in the main cache may be invalidated by another thread and reloaded from memory into the assist cache.

Data is only stored to memory when it is flushed from the main cache due to a collision or due to another processor loading it. This happens when the least significant 18 bits of the address of a data item being written to the main cache is identical to the least significant 18 bits of a data item already resident in the main cache. In this case the resident item is replaced by the new item in the cache. If the resident cache line has been modified, it is written to memory as part of the replacement process.

If an item being stored does not reside in either cache, the appropriate cache line is reloaded into the assist cache (as with a load) and the data is written there.

### Prefetching

When a cache line is fetched or accessed on an SPP1200 Series machine, hardware in the CPU can predict which cache line is likely to be needed next based on loop stride and direction and, if the predicted line is within the same page, *prefetch* it. This means that on an initial fetch from memory, two cache lines are actually fetched—the one containing the data needed for the current load or store, and the one containing data likely to be fetched next, even if that data is not residing in an adjoining cache line.

A *prefetched cache line* is, by definition, an as-yet-unused cache line residing in the cache as a result of another line being fetched. It is possible that a cache line will be prefetched into the assist cache and eventually be displaced to the main cache without ever being used; this is the only way that unused data can make it into the main cache.

Prefetching takes place every time data is fetched from memory and every time a prefetched cache line is accessed in the assist cache. When an encached prefetched line is accessed, if the cache line likely to be needed next is not already encached, it is prefetched. If the CPU cannot predict what cache line will be needed next but knows the direction in which data is being accessed, it generally prefetches the cache line that is contiguous to the fetched line in the appropriate direction. The only time no prefetching occurs is when both the requested cache line and the predicted next cache line are already encached, or the prefetch crosses a page boundary.

The actual prefetching operation does not execute simultaneously with the fetch, but immediately thereafter, and it effectively executes simultaneously with the instructions executing on the CPU. Up to four prefetches can be outstanding.

## Cache thrashing—SPP1000 Series

SPP1000 Series machines use a 1-Mbyte direct-mapped data cache. In a direct-mapped cache, unique data objects can have identical cache addresses. On SPP1000 Series machines, objects whose virtual addresses are identical in their least-significant 20 bits have identical cache addresses. If such data objects are referenced from within the same loop, they will overwrite each other in the cache, generally resulting in constant cache misses, and causing unnecessary and costly main-memory accesses. This condition is known as *cache thrashing*.

Cache thrashing can become a problem on SPP1000 Series machines when two encachable data objects are exactly 1 Mbyte apart in memory. To eliminate the problem, you must ensure that your data is not spaced this way.

Consider the following Fortran example.

```
REAL*8 ORIG(65536),NEW(65536),DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
DO I = 1, N
    NEW(I) = ORIG(I) + DISP(I)
ENDDO
```

In this example, the arrays ORIG and DISP will overwrite each other in a 1 Mbyte cache. Because the arrays are in a common block, we know that they will be allocated in contiguous memory in the order shown. Each array element occupies 8 bytes, so each array occupies .5 Mbyte ( $8 \times 65536 = 524288$  bytes); therefore arrays ORIG and DISP are exactly 1 Mbyte apart in memory, and all their elements have identical cache addresses. The layout of the arrays in memory and in the data cache is shown in Figure 4.

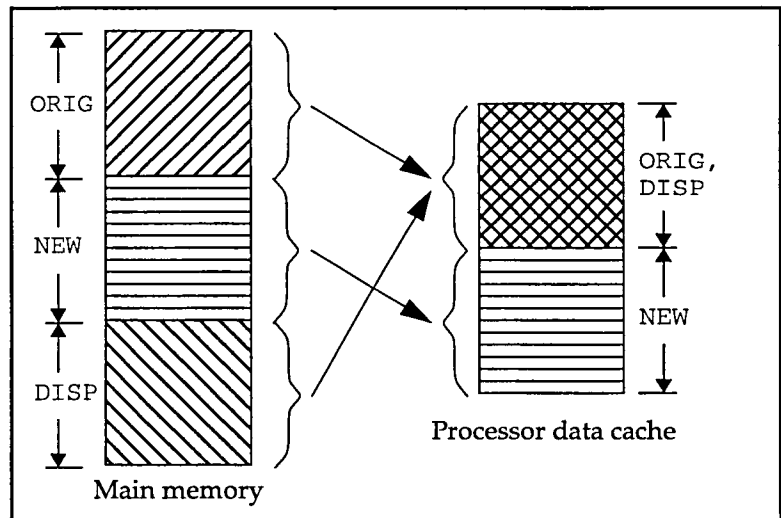


Figure 4 Array layouts—cache-thrashing

When the addition in the body of the loop executes, the current elements of both *ORIG* and *DISP* must be fetched from main memory into the cache. Since these elements have identical cache addresses, whichever is fetched last will overwrite the first. Remember that processor cache data is fetched 32 bytes at a time; to efficiently execute a loop such as this, the unused elements in the fetched cache line (3 extra *REAL\*8* elements are fetched in this case) must remain encached until they can be used in subsequent iterations of the loop. Because *ORIG* and *DISP* thrash each other, this reuse is never possible; every cache line of *ORIG* that is fetched is overwritten by the cache line of *DISP* that is subsequently fetched, and vice versa. The cache line is overwritten on every iteration; typically in a loop like this, it would not be overwritten until all of its elements were used.

Since main memory accesses take substantially longer than cache accesses, this severely degrades performance. Even if the overwriting involved the *NEW* array, which is stored rather than loaded on each iteration, thrashing would occur, because stores overwrite entire cache lines the same way loads do.

But the problem is easily fixed by increasing the distance between the arrays. This can be done by either increasing the array sizes or inserting a padding array.

The following example illustrates the padding approach.

```
REAL*8 ORIG(65536),NEW(65536),P(8),DISP(65536)
COMMON /BLK1/ ORIG, NEW, P, DISP
.
.
.
```

Here, the array P(8) moves DISP 64 bytes further from ORIG in memory. Now no two elements of the same index share an SPP1000 Series cache address, and for the given loop, this postpones cache overwriting until the entire current cache line is completely exploited. P is 8 elements, or 64 bytes, which prevents CTIcache as well as processor cache thrashing; a size of 32 bytes would work to prevent processor cache thrashing, but there is a slim chance CTIcache thrashing could still occur since a CTIcache line is 64 bytes.

The alternate approach involves increasing the size of ORIG or NEW by 8 elements (64 bytes), as shown in the following example.

```
REAL*8 ORIG(65536),NEW(65544),DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
```

Here, NEW has been increased by 8 elements, providing the padding necessary to prevent ORIG from sharing cache addresses with DISP. Figure 5 shows how both solutions prevent thrashing. Note that the pad never gets encached, whether it consists of the P array or an extended NEW array, because it is never referenced. If it was referenced, it would not solve the thrashing problem.

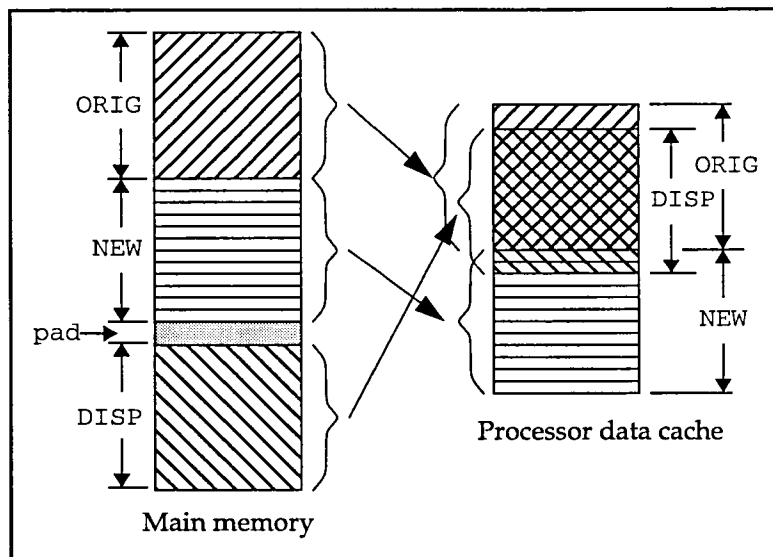


Figure 5 Array layouts, non-thrashing

It is important to note that this is a highly simplified, worst-case example. On SPP1000 Series machines, thrashing can happen any time two data items that are referenced in the same loop are an integral multiple of 1 Mbyte apart in virtual memory. This can happen with data that is not stored in common, in which case it is much more difficult to see, as such data can be stored noncontiguously and may be intermixed with completely unrelated data items. The loop blocking optimization (described in Chapter 3, "Compiler optimizations") will eliminate thrashing from certain nested loops, but not from all loops. Declaring arrays with dimensions that are not powers of two can help, but will not necessarily eliminate the problem completely. Using common blocks in Fortran can also help because it allows you to accurately measure distances between data items, making thrashing problems easier to spot before they happen.

### Cache thrashing—SPP1200 Series

The thrashing problem is so well-mitigated by the SPP1200 Series' assist cache that it is impossible to contrive a simple example to demonstrate it.

There are two sets of conditions that may cause cache thrashing on SPP1200 Series machines. In the first, two thrashing data items, which we will arbitrarily call  $x$  and  $y$ , must be accessed in the following way:

- $x$  and  $y$  are encached into the assist cache consecutively;  $y$  therefore immediately follows  $x$  into the cache.

- $x$  and  $y$ , having entered the cache consecutively, move through the queue consecutively, so that when  $x$  is displaced from the main cache,  $y$  is the next cache entry to be displaced.
- $y$  displaces  $x$  from the main cache. To do this, the addresses of  $x$  and  $y$  must be identical in their lowest order 18 bits. So when  $y$  is displaced from the assist cache to the main cache immediately after  $x$  is displaced,  $y$  displaces  $x$  from the main cache. If this condition is not met, both items will always be available in the main cache after being displaced from the assist cache.
- In order to adversely affect performance, no reuse of  $x$  or  $y$ 's cache line can occur while it resides in either cache, and the program must reference some element of  $x$ 's cache line after  $y$  displaces it from the main cache, causing the cycle to repeat.

The other conditions that may cause cache thrashing on SPP1200 Series machines require  $x$  and  $y$  to be accessed in the following way:

- They cyclically displace each other from the assist cache, with no reuse of either item's cache line between displacements. To do this, the loop must be written so that  $y$  is referenced at the exact instant when  $x$  is the next item to be displaced from the assist cache. This condition is difficult to envision because the algorithm that determines when an assist-cache entry must be displaced depends on many variable runtime conditions.
- They displace each other from the main cache. To do this, the addresses of  $x$  and  $y$  must be identical in their lowest order 18 bits. If this condition is not met, both items will always be available in the main cache after being displaced from the assist cache.
- In order to adversely affect performance, no reuse on  $x$ 's cache line can be exploited before  $x$  is displaced by  $y$  in either cache. This means that during all the references that must occur between the reference of  $x$  and the reference of  $y$  in order for  $y$  to overwrite  $x$  in the assist cache, no reuse on  $x$ 's cache line can occur. Similarly, if  $x$  resides in the main cache, no reuse on its cache line can occur before it is overwritten when  $y$  is displaced from the assist cache by some other data item.

Obviously, it is hard to imagine a loop that satisfies either set of conditions for cache thrashing on SPP1200 Series machines. If you experience performance problems in a loop and find that it is accessing data as discussed in the preceding conditions for thrashing, try inserting a 64 byte pad between the arrays you suspect are thrashing using the method described in the "Cache thrashing—SPP1000 Series" section. This should prevent main cache thrashing and alleviate the problem.

## Cache use analysis

CXpa, the Convex visual profiler, can be used to analyze cache performance. CXpa's many features include the ability to analyze the following data:

- Locally resolved (CTI not used) cache miss counts and latency
- Remotely resolved (CTI used) cache miss counts and latency
- Locally and remotely resolved (total) cache miss counts and latency
- Average cache miss latency

CXpa is an optional product. For more information about CXpa, refer to the *CXpa Reference for Exemplar Systems* (DSW-605) or contact your Convex sales representative.

## Data alignment

Aligning data addresses on cache line boundaries allows for efficient data reuse in loops, especially when the compiler performs the blocking optimization (refer to Chapter 3, "Compiler optimizations"). The following data is aligned on CTIcache boundaries automatically:

- Data declared as part of a Fortran COMMON block
- Data allocated using a Fortran ALLOCATE or C malloc or memory\_class\_malloc statement
- Uninitialized data of the external storage class in C.

Only the first item in a list of data objects appearing in any of these statements is aligned. To make most efficient use of available memory, the total size in bytes of any array appearing in one of these statements should be integrally divisible by 64 (the size of a CTIcache line in bytes). Sizing your arrays this way will prevent data following the first array from becoming misaligned. Scalar variables should be listed after arrays, and ordered from longest data type to shortest (e.g. REAL\*8 scalars should precede REAL\*4's).

## Note

**Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.**

You can force CTIcache boundary alignment for specific scalar variables or arrays by using the `align_cti` directive or pragma. The Fortran directive has the following form:

```
C$DIR ALIGN_CTI (namelist)
```

In C it has the following form:

```
#pragma _CNX align_cti(namelist)
```

Where *namelist* is a list of arrays and/or scalars that will be aligned on CTIcache boundaries.

You can specify CTIcache boundary alignment for all arrays (but not scalars) in your program by using the `-align cti` compiler option.

---

## Interleaving

Physical pages are interleaved across the 8 banks of a hypernode by CTIcache lines. Contiguous CTIcache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown in Figure 6.

Each memory block shown in Figure 6 is located on a separate functional block within the hypernode.

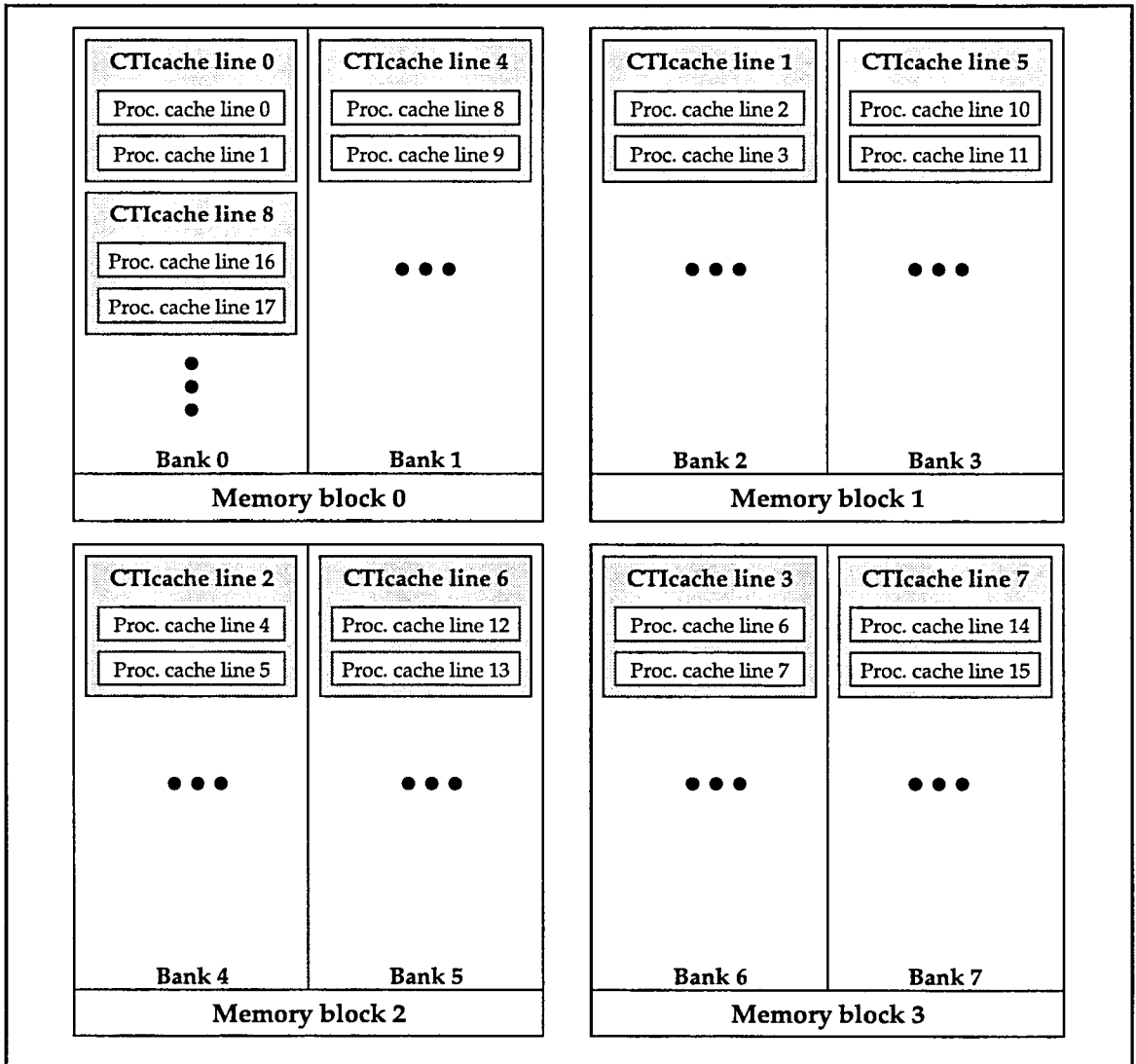


Figure 6 SPP Series memory interleaving

Interleaving speeds memory accesses by allowing several processors to access contiguous data simultaneously. This is extremely beneficial when a loop that manipulates arrays is split among many processors; in the best case, threads will access data in patterns with no bank contention. Even in the worst case, where each thread initially needs the same data from the same bank, after the initial contention delay the accesses will be spread out among the banks.

## Interleaving example

The following example illustrates a nested loop that accesses memory with very little contention. This example is greatly simplified for illustrative purposes, but the concepts apply to arrays of any size.

```
REAL*8 A(12,12), B(12,12)
...
DO J = 1, N
  DO I = 1, N
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Assume that arrays A and B are stored contiguously in main memory, with A starting in bank 0, CTIcache line 0, processor cache line 0, as shown in Figure 7.

Assume the Convex SPP Series Fortran compiler parallelizes the J loop to run on as many processors as are available in the subcomplex (up to N). Assuming N=12 and there are four processors available when the program is run, the J loop could be divided into four new loops, each with 3 iterations. Each new loop would run to completion on a separate processor. We'll refer to these four processors as CPU0 through CPU3.

## Note

This example is designed to simplify illustration. In reality, the dynamic selection optimization (discussed in Chapter 3, "Compiler optimizations") would, given the iteration count and available number of processors described, cause this loop to run serially. The overhead of going parallel would outweigh the benefits.

In order to execute the body of the I loop, A and B must be fetched from memory and encached. Each of the four processors running the J loop will attempt to fetch its portion of the arrays, most likely simultaneously.

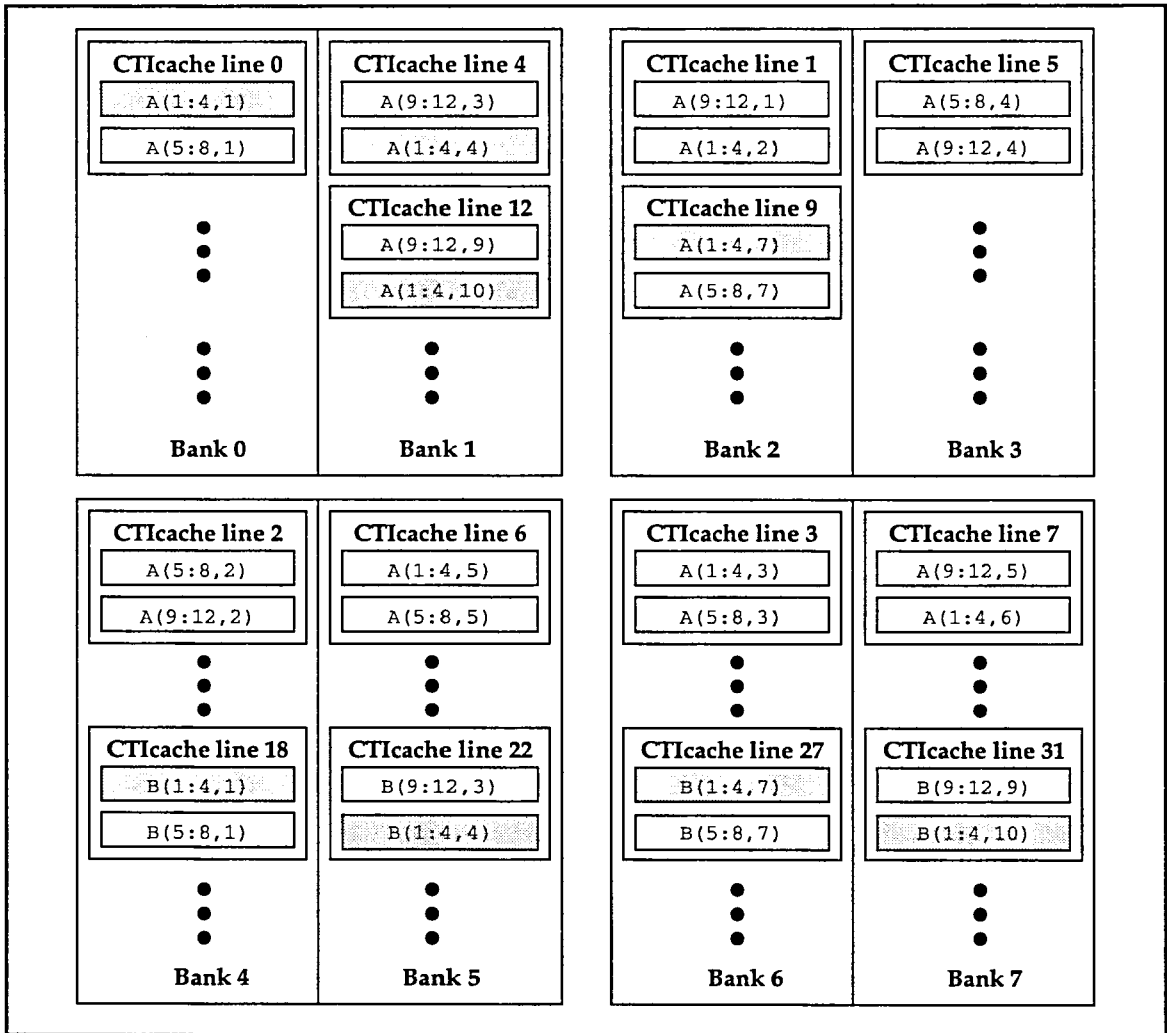


Figure 7 Interleaving of arrays A, B, and C

CPU0 needs  $A(1:12, 1:3)$  and  $B(1:12, 1:3)$ ; CPU1 needs  $A(1:12, 4:6)$  and  $B(1:12, 4:6)$ ; CPU2 needs  $A(1:12, 7:9)$  and  $B(1:12, 7:9)$ ; CPU3 needs  $A(1:12, 10:12)$  and  $B(1:12, 10:12)$ . This means CPU0 will attempt to read arrays A and B starting at elements (1, 1), CPU1 will attempt to start at elements (1, 4) and so on. For clarity, Figure 7 shows the first 8 CTIcache lines consecutively; after these, only the initial cache lines for each processor are shown. Each processor's initial cache line is shaded.

Without interleaving, all four CPUs would request their arrays simultaneously from the same bank. Arbitration logic would order the CPUs; whichever ended up going first would get its

requested cache line while the second experienced a ~255 ns delay (about 26 processor clocks on an SPP1000 system, 31 processor clocks on an SPP1200 system), the third a ~510 ns delay, and the fourth a ~765 ns delay. Before the second CPU had even received its cache line, the first would likely be back in the queue waiting for its next cache line.

Interleaving helps to eliminate such contention.  $A(1, 1)$ , which is the first element of the chunk needed by CPU0, is on cache line 0 in bank 0;  $A(1, 4)$ , the first element needed by CPU1, is on cache line 4 in bank 1;  $A(1, 7)$ , the first element needed by CPU2, is on cache line 9 in bank 2, and  $A(1, 10)$ , the first element needed by CPU3, is on cache line 12 in bank 1. Contention only exists between CPU1 and CPU3, and after one of these CPUs gets its cache line and moves on, it will proceed at full speed with the other CPU 28 clocks behind it. In other words, after the initial access, the contention will cease. Contention may resurface occasionally as the processors make their way through the data, but the resulting delays are minimal compared to what could be expected without interleaving.

The initial chunks of B are even more favorably distributed, in banks 4, 5, 6, and 7, respectively. No initial contention exists for B.

---

## Subcomplexes

On Exemplar systems, processes run on virtual machines called subcomplexes, which are arbitrary collections of processors. Subcomplexes are highly configurable, and configuration is done using the Convex Subcomplex Manager. For more information on the Subcomplex Manager, refer to the *SPP-UX System Administrator's Guide*.

Subcomplexes allow the system administrator to tailor processors and memory to your specific application needs, making the most efficient use of your Exemplar system resources. For example, a nonparallel or non-time-critical application can be allocated a single processor; an application containing a lot of fine-grain parallelism can be allocated many processors, and an application requiring large amounts of memory can be allocated processors on several hypernodes. Because subcomplexes can be managed without rebooting, you can adjust configurations to automatically exploit periodic changes in system usage such as late-night or weekend idle time.

---

### Physical configuration

Subcomplexes can consist of from one processor to the total number installed on the machine. Hypernodes can be split among as many subcomplexes as there are processors in the hypernode, and subcomplexes can be subsets or supersets of hypernodes. Processors can only belong to one subcomplex at a time, and every processor must belong to a subcomplex.

Figure 8 shows an 8-hypernode, 64-processor system split into 5 subcomplexes.

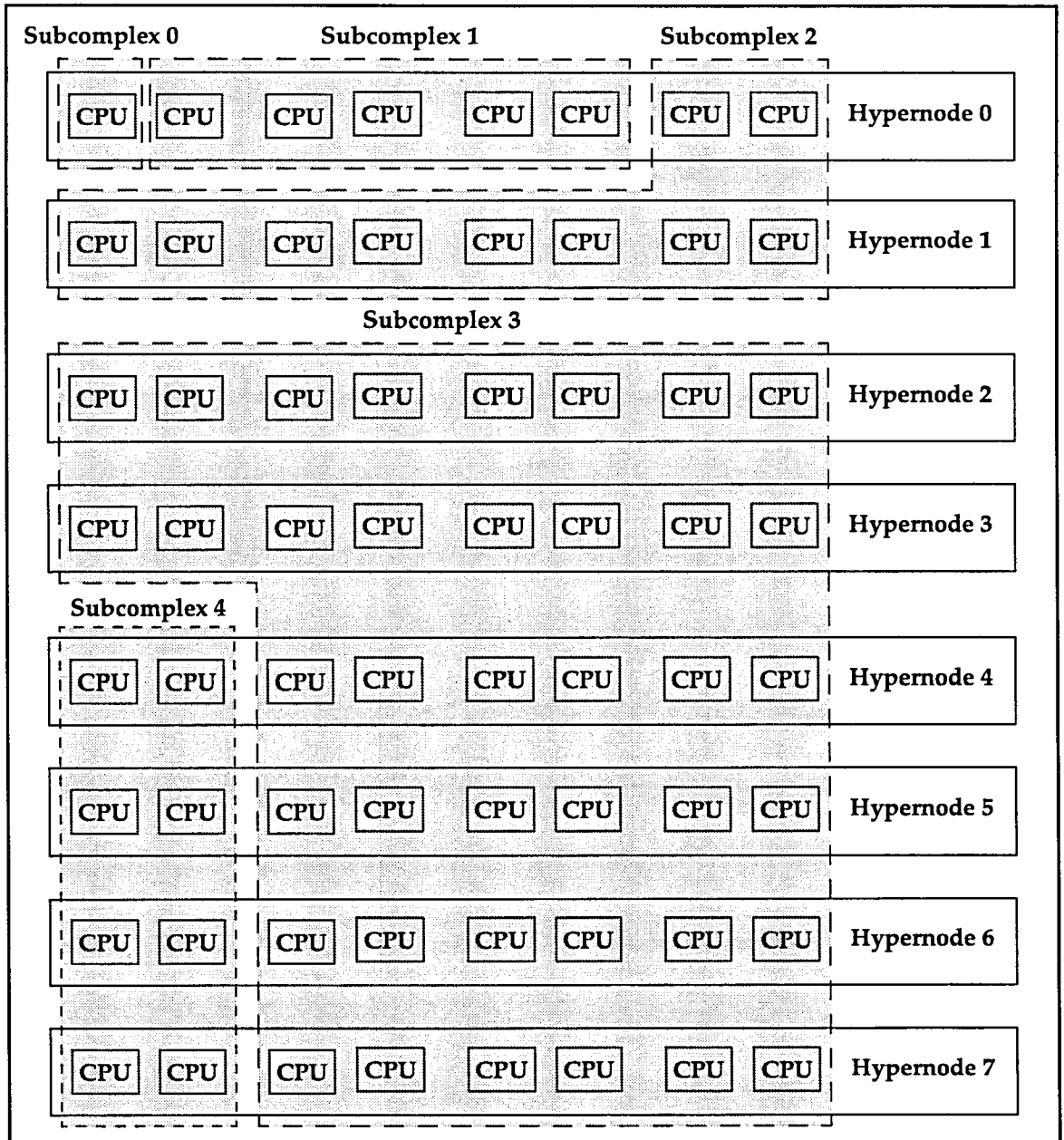


Figure 8 Hypothetical subcomplex configurations

Processors can be added to or removed from a subcomplex at any time; depending on the global memory requirements of the hypernodes making up the subcomplex, the subcomplex may or may not need to go idle. Idling a subcomplex on which processes are running will cause the processes to go idle; activating the subcomplex will automatically activate the idle threads.

Subcomplex memory can also be configured using the Subcomplex Manager. Global memory can be allocated when the subcomplex is idle, in 16Mbyte increments, up to the total amount available, from any hypernode which has one or more processors in the subcomplex. On SPP1200 systems, CTIcache memory can be configured at anytime, and different hypernodes can reserve different amounts of CTIcache. On SPP1000 systems, CTIcache must be configured at system start up, and the CTIcache size must be uniform across all hypernodes.

Also configurable are subcomplex permissions, which are similar to file permissions and can be set at the user, group, and world level. Read permissions on a subcomplex allow you to read the subcomplex configuration; write permissions allow you to change the subcomplex's job scheduling policies, such as process execution priority; execute permissions allow you to run a process on the subcomplex. Only the system administrator can configure or reconfigure a subcomplex.

---

## Subcomplex memory

GSM is global to a subcomplex, and processes run entirely within their assigned subcomplex. For a program to run on more than one subcomplex, message passing via sockets (not the CTI rings) must be used to communicate between the subcomplexes; this is much slower than interhypernode communication within a subcomplex.

The virtual address spaces accessible from within a subcomplex are unique; this prevents processes running on one subcomplex from accessing memory on another subcomplex. Furthermore, the virtual address spaces accessible to each process running on a subcomplex are unique; each process is assigned a 4Gbyte virtual space, and processors cannot access each other's memory.

When two or more subcomplexes share a single hypernode, the physical memory pages assigned as global memory for each subcomplex are unique, preventing contention over global memory. However, the physical pages associated with hypernode-local and CTIcache memory are shared among multiple subcomplexes running on one hypernode; this can result in contention over these resources.



This chapter discusses the various optimization options available for use with the SPP Series Fortran and C compilers and provides detailed explanations of the optimizations performed under each option.

## Optimization options

Five optimization options are available for use with the SPP Series C and Fortran compilers. These options have identical names and perform identical optimizations regardless of which compiler you're using. They are specified on the compiler command line along with any other options you wish to use. SPP Series compiler optimization levels are summarized in Table 1.

**Table 1** SPP Series C and Fortran optimization options

Option	Description
-no	Machine instruction level scalar optimizations.
-O0	Basic block level scalar optimizations.
-O1	Program unit level scalar optimizations and global register allocation.
-O2	Global instruction scheduling and data localization optimizations. This option is the default.
-O3	Parallel optimizations.

These options are cumulative; each option retains the optimizations of the previous option. For example, entering the following command line:

```
% fc -O2 foo.f
```

would compile the Fortran program `foo.f` with all -O2, -O1, -O0, and -no level optimizations shown in Table 1.

In addition to these options, the `-noautopar` option is available for use with the `-O3` option. `-noautopar` will cause the compiler to parallelize only those loops that are immediately preceded by `loop_parallel` or `prefer_parallel` directives or pragmas; for more information, refer to Chapter 4, "Basic shared memory programming."

---

## **-no level optimizations**

At optimization level `-no`, the compiler performs scalar optimizations that span no more than a single source statement. These optimizations create object code that fully uses the scalar features of the PA-RISC architecture.

---

### **Instruction scheduling**

Instruction scheduling rearranges machine instructions to use the computer's functional units most effectively. Each CPU on an SPP Series system has multiple functional units on which operations execute simultaneously.

At optimization level `-no`, the compiler rearranges the machine instructions generated by no more than a single source statement to maximize use of the functional units.

Concurrent execution of machine instructions on multiple functional units, within a single processor, is distinct from parallel processing, which occurs on multiple processors.

---

### **Span-dependent instructions**

There are several ways to specify a branch in the PA-RISC machine language; short branches can be more efficiently executed than long branches. The SPP Series compilers automatically generate branches using the most efficient and appropriate branching techniques.

For more information on branch instructions, see the Hewlett Packard *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*.

---

### **Register allocation**

The SPP Series compilers use a technique for allocating registers that fully exploits the PA-RISC register set. This allows grouping of register loads and concurrent execution of instructions (*pipelining*), and reduces register conflicts.

## Tree-height reduction

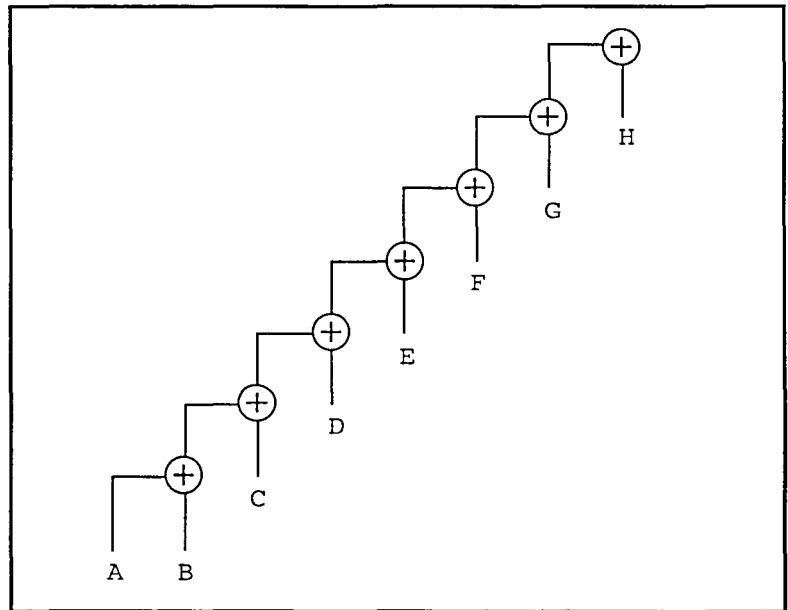
The compiler represents expressions internally as trees. When they involve floating point numbers, these trees are optimized by *tree-height reduction* or *balancing*. For example, consider this Fortran expression involving REAL variables:

$A + B + C + D + E + F + G + H$

The expression can be evaluated as follows:

$(((((A + B) + C) + D) + E) + F) + G) + H)$

$(A+B)$  is evaluated first, the result is added to C, and so on until the expression is fully evaluated. Figure 9 shows how the compiler represents this order internally.



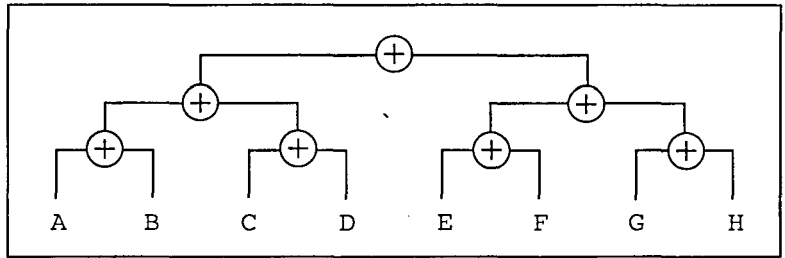
**Figure 9** Unbalanced tree representation

The PA-RISC processor's add functional unit takes two clocks to complete an add, but pipelining allows it to issue one add result per clock if the pipeline is kept full. The unit does not chain, so the result of an add in progress cannot be used in the next add. Since each addition in this example depends on the result of the addition to the left, pipelining in the add functional unit cannot be exploited; it can then only issue one result every two clocks.

Another way to evaluate the expression is

$((A + B) + (C + D)) + ((E + F) + (G + H))$

This is represented internally as shown in Figure 10.



**Figure 10** Balanced tree representation

In Figure 9, the depth of the tree is seven; in Figure 10, the depth of the tree is three. The code generated for the tree in Figure 9 executes more slowly than that generated for the tree in Figure 10.

This is because none of the four additions in the innermost parentheses (represented as leaf nodes on the tree in ) requires the result of another addition. The additions can therefore be fed to the add functional unit so that they fill its pipeline, allowing it to issue one result per clock after the first add and up until the last.  $(A+B)$  is evaluated on the first clock, followed by  $(C+D)$  on the second; on the third,  $(A+B)$  is done and the next add is starting. This pattern continues until the expression is fully evaluated.

The deeper the tree representing the expression, the more time is required to evaluate the expression. If a particular evaluation order is not specified with parentheses, the compilers choose one that minimizes the depth of the expression and maximizes instruction pipelining, while maintaining the arithmetically correct evaluation order. Because the compilers choose evaluation order to ensure the most efficient execution, you can write expressions in any order; if your expression requires a certain order, be sure to indicate it with parentheses.

---

### Short-circuit evaluation of conditionals in Fortran

Short-circuiting the evaluation of conditionals is ANSI standard behavior in C, but Convex Fortran also performs this optimization.

Short circuiting increases the efficiency of IF statements by skipping irrelevant tests when logical operators are involved in the conditional. Convex Fortran short-circuits evaluation of IF statements that contain `.AND.` and `.OR.` operators that have logical operands and are used in a logical context. Take, for example, the following Fortran IF statement:

```
IF ((A .EQ. B) .OR. F(G)) THEN
```

If (A .EQ. B) evaluates to true, the evaluation of F(G) is skipped, and the THEN portion of the statement is evaluated.

Similarly, given the Fortran code

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if (A .EQ. B) evaluates to false, the evaluation of F(G) and the THEN portion of the statement is skipped.

Short-circuit evaluation works with all types of IF statements (arithmetic, logical, and block). Performing arithmetic (+, -, \*, /) on a logical expression disables short circuiting within that expression. Logical-valued expressions used as arguments to function calls within an IF statement's conditional expression may not be short circuited. Note that the binary operators .EQ., .NE., .LT., .LE., .GT., and .GE. always produce a logical result.

The compiler short-circuits the evaluation of conditionals by default. You can disable short-circuiting in Fortran by specifying the `-nosc` flag on the compiler command line.

---

## Data alignment

To provide more efficient access to data, the compiler automatically aligns data objects to their natural boundaries in memory. This means that a data object's address is integrally divisible by the length of its data type; e.g. REAL\*8 objects have addresses integrally divisible by 8 bytes.

**Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.**

To ensure the efficient layout of such aligned data in memory, you should declare scalar variables in order from longest to shortest data length. This minimizes the amount of padding the compiler has to do to get the data onto its natural boundary. Consider the following Fortran example:

```
C      CAUTION: POORLY ORDERED DATA FOLLOWS:
      LOGICAL*2  BOOL
      INTEGER*8  A, B
      REAL*4     C
      REAL*8     D
```

Here, the compiler must insert 6 blank bytes after `BOOL` in order to correctly align `A`, and it must insert 4 blank bytes after `C` to correctly align `D`.

## Note

The same data is more efficiently ordered as shown in the following example:

```
C      PROPERLY ORDERED DATA FOLLOWS:
      INTEGER*8 A, B
      REAL*8 D
      REAL*4 C
      LOGICAL*2 BOOL
```

Natural boundary alignment is the default on SPP Series Convex compilers. Traditional C Series-like tight packing of variables can be specified using the `-align cseries` option as described in Appendix B.

Natural boundary alignment is performed on all data. It should not be confused with CTIcache line boundary alignment, which is performed as described in the "Data alignment" section of Chapter 2, "Architecture overview." Also discussed in Chapter 2 are the `align_cti` directive and pragma and the `-align cti` compiler option, all of which facilitate CTIcache line boundary alignment.

---

## **-O0 Level optimizations**

At optimization level `-O0`, the compiler performs scalar optimizations within a basic block; a basic block is a sequence of statements with only one entry point and one exit. Basic blocks may end with a branch, but they cannot contain branches. The compiler also continues to perform the optimizations performed at `-no`.

---

### **Instruction scheduling**

At optimization level `-O0` and above, instructions from *multiple* statements, as well as those from single statements, are scheduled as a group.

---

### **Redundant-assignment elimination**

Redundant-assignment elimination removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated. The code in

the following Fortran example contains a redundant assignment,  $X=Y+C$ , which the compiler removes.

Original code	Optimized code
$X = Y + C$	!(statement eliminated)
!(X not used)	.
.	.
.	.
$X = 3.1416$	$X = 3.1416$
.	.
.	.
.	.
$Y = (X + 7) * 2.15$	$Y = (X + 7) * 2.15$

---

### Assignment substitution

Assignment substitution eliminates redundant loads. The compiler stores the value assigned to a variable in a register and references the register in subsequent references to the variable. A Fortran example appears below.

Original code	Optimized code
$X = Y + C$	$REG = Y + C$
$X = X * 4.4$	$REG = REG * 4.4$
$T = X * B + 12.4$	$T = REG * B + 12.4$
$X = 4.179$	$X = 4.179$

After the machine instructions for the first statement execute, the value of  $Y+C$  remains in a register. The compiler replaces subsequent references to  $X$  with references to this register until the value of  $X$  changes or until the end of the basic block is reached. This optimization eliminates repeated loading and storing of  $X$  into a register, which increases performance and provides opportunities for further optimization. In this example, assignment substitution makes the first assignment to  $X$  redundant, so the compiler eliminates the assignment.

Because the compiler substitutes assignments, you rarely need to optimize a program by replacing a variable reference with a constant in the source code.

---

## Common-subexpression elimination

The compiler recognizes subexpressions that repeat within a basic block. The compiler retains the value of the subexpression in a register, which eliminates redundant computations and register loads. For example, the compiler recognizes  $B+C$  as a common subexpression of  $A+B+C+D$  and  $B+E+C$  and calculates the subexpression only once.

The compiler also eliminates redundant array address calculations. As with assignment substitution, you do not need to manually create a temporary variable in which to store the value of a common subexpression. The compiler performs that function automatically.

---

## Redundant-use elimination

This optimization is a special case of common subexpression elimination where the subexpression is a variable. The compiler detects multiple references to a variable between assignments and retains the value of the variable in a register. This action helps eliminate redundant register loads.

---

## Constant propagation and folding

After assigning a constant to a variable, the compiler replaces subsequent references to the variable with the constant. For example, if you write  $X=5$ , the compiler replaces  $X$  with 5 within that basic block or until a new value is assigned to the variable. This is known as *constant propagation*, which is a form of assignment substitution.

A Fortran example of constant propagation and folding follows.

Original code	Optimized code
I = 5	I = 5
J = 0	!(assignment eliminated)
.	.
.	.
.	.
J = J + 2	J = 2
.	.
.	.
.	.
K = K + I * J	K = K + 10

The compiler also replaces operations on constants with the result of the operation. This is known as *constant folding*. For example, it replaces  $Y=5+7$  with  $Y=12$ . It then propagates the constant value to replace future references to  $Y$  within the basic block. The Fortran compiler also propagates and folds values assigned to names in `PARAMETER` statements.

The compiler folds the most frequently used Fortran intrinsics and C library routines when they are applied to constant arguments. For example,  $\text{SIN}(0.0)$  becomes  $0.0$ . The compiler also folds exponentiation involving constants. For example,  $3^{**}3$  becomes  $27$ .

The compiler type-converts constants, if necessary, before propagating and folding them. If a Fortran program contains the expression  $X=1$ , where  $X$  is `REAL`, the compiler converts  $1$  to  $1.0$  before propagating it.

If an integer overflow occurs as a result of constant folding in Fortran, the compiler reports "Integer constant truncation." To see such a message in C, you must supply the `-d integer_overflow=e` option on the `cc` command line. If a floating-point overflow occurs, both compilers report "Real constant either too large or too small." Floating-point underflow always results in zero. If any of these messages or conditions occur, eliminate the offending operation or bring the value of the constant within acceptable bounds.

---

## Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown in the following Fortran examples.

Original expression	Optimized expression
$X + 0$	$X$
$X * 1$	$X$
$X * 0$	$0$
$K .AND. -1$	$K$
$K .AND. 0$	$0$
$K .OR. -1$	$-1$
$K .OR. 0$	$K$
$-1 * X$	$-X$
$X - X$	$0$
$X / -1$	$-X$
$(-1) ** K$	$1 - ((K .AND. 1) * 2)$
$X ** 0.5$	$SQRT(X)$
$X ** 0$	$1$
$1 ** X$	$1$
$X / X$	$1$
$0 - X$	$-X$
$0 / X$	$0$
$SIN(X) * COS(X)$	$0.5 * SIN(2X)$
$SIN(X) / COS(X)$	$TAN(X)$

The compiler performs obvious variations of these operations for the commutative operators. For example, in Fortran the compiler converts  $X + (0 + Y)$  to  $X + Y$ .

---

## **-O1 Level optimizations**

At optimization level -O1, global optimization is done across a group of basic blocks but within a single procedure. The -O1 option performs global, basic-block, and machine instruction level optimizations.

---

### **Constant propagation and folding**

Propagating and folding constants at the procedure level is analogous to performing the same operations at the basic-block level. The scope of the optimization is now an entire procedure.

A Fortran example of constant propagation and folding follows.

<b>Original code</b>	<b>Optimized code</b>
INTEGER A, B, C	INTEGER A, B, C
A = 5	A = 5
B = 15	B = 15
READ *, I	READ *, I
IF (I) 10, 10, 15	IF (I) 10, 10, 15
10 A = 6	10 A = 6
C = A	C = 6 !A=6
GOTO 20	GOTO 20
15 C = A + B	15 C = 20 !A=5, B=15
GOTO 25	GOTO 25
20 B = A + C	20 B = 12 !A=6, C=6
GOTO 30	GOTO 30
25 B = A + B + C	25 B = 40 !A=5, B=15, C=20
30 PRINT *, A, B, C	30 PRINT *, A, B, C
END	END

The compiler propagates and folds constants globally at optimization level -O1 and higher, which eliminates the need to propagate constants by hand in programs compiled at these levels.

---

## Redundant-assignment elimination

At optimization level -O1, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. The following Fortran example shows how the compiler eliminates redundant assignments to the variable A.

### Original code

```
SUBROUTINE FOO
  INTEGER A
  .
  .
  .
  X = Y * Z
  A = Y**3
  ASSIGN 10 TO B
  .
  .
  .
  IF (A .GT. 0) THEN
    .
    .
    .
    A = X * Y + 3.1416
    GOTO B
  ELSE
    .
    .
    .
    X = (X + 7) * Z + 3.1416 ! X USED LATER
  ENDIF
  .
  .
  .
C  A is not used later in this routine
  END
```

As shown in the optimized code that follows, the Fortran compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and common variables.

### Optimized code

```
SUBROUTINE FOO
INTEGER A
.
.
.
X = Y * Z
A = Y**3
ASSIGN 10 TO B
.
.
.
IF (A .GT. 0) THEN
.
. ! ASSIGNMENT TO A REMOVED
.
GOTO B
ELSE
.
.
.
X = (X + 7) * Z + 3.1416 ! X USED LATER
ENDIF
.
.
.
END
```

If the right side of a redundant assignment statement contains a procedure call, the compiler eliminates the assignment and retains the call, as in the following Fortran example.

#### Original code

```
SUBROUTINE FOO
```

```
.
```

```
.
```

```
.
```

```
I = INTFUN(X)
```

```
.
```

```
.
```

```
.
```

#### Optimized code

```
SUBROUTINE FOO
```

```
.
```

```
.
```

```
.
```

```
<NULL> = INTFUN(X)
```

```
.
```

```
.
```

```
.
```

Comment: I not used

If a C or Fortran function appearing in an assignment has no side effects, the compiler can eliminate the function call, as well as the assignment, improving efficiency. A procedure free of side effects if it

- Doesn't modify the value of an argument
- Doesn't modify the value of a global or Fortran common variable
- Doesn't modify the value of local static variables used on subsequent calls
- Doesn't perform input or output
- Doesn't call another procedure that has side effects

Existing procedure compilers cannot automatically determine whether a side effect exists. The Convex Fortran and C compilers eliminate procedure calls only if you explicitly request this behavior with the `NO_SIDE_EFFECTS` directive or pragma.

The form of this Fortran directive is

```
C$DIR NO_SIDE_EFFECTS (func_list)
```

The form of the C pragma is

```
#pragma _CNX no_side_effects (func_list)
```

where *func\_list* is a list of procedure names separated by commas. The directive must precede the procedure call that does not contain side effects.

---

## Caution

---

Do not use the `NO_SIDE_EFFECTS` directive or pragma on a call to a procedure that

- Changes the value of an argument
- Changes the value of a `COMMON` or global variable
- Performs input or output
- Changes the value of a static local variable in C
- Changes the value of a `SAVED` local variable in Fortran
- Calls another procedure that performs one of these operations

For more information about the `NO_SIDE_EFFECTS` directive, see Appendix A, "Compiler directives."

The Convex Application Compiler is capable of recognizing side effects in procedure calls. Refer to the *Application Compiler User's Guide* for more information.

---

## Dead-code elimination

If, as a result of constant propagation and folding, the compiler can fold an arithmetic or logical expression in a Fortran IF statement to `.TRUE.` or `.FALSE.`, or if the expression can be folded to a constant in C, the compiler eliminates any unreachable code that results.

---

## Copy propagation

The compiler can replace a variable with another variable to which it has been equated. This is called *copy propagation*. For example, after evaluating the statement `X=Y`, the compiler replaces later occurrences of `X` with `Y`.

In the following Fortran example, if the compiler determines that X and Y are unchanged between the assignment and the reference, it replaces X with Y.

```
X = Y
.
.
.
W = Z - X
```

becomes

```
.
.
.
W = Z - Y
```

---

## Common subexpression elimination

The compiler eliminates common subexpressions at the global level. The compiler retains the value of the common subexpression in a register if one is available; otherwise, the compiler assigns the value to a temporary variable. The compiler then replaces subsequent occurrences of the common subexpression with references to the register or temporary variable.

In the following Fortran example, the compiler determines that the subexpression must be calculated whether the condition associated with the IF statement evaluates to `.TRUE.` or `.FALSE.`

```
SUBROUTINE GCSE2
.
.
.
IF (K .LT. L) THEN
  A = (C * 4) / -(J * B + SQRT(C))
ELSE
  E = (E * 4) / -(J * B + SQRT(C))
ENDIF
F = (B * 4) / -(J * B + SQRT(C))
.
.
.
END
```

The compiler saves the value of the common subexpression in the temporary variable T1 and uses the variable to compute the value for assignment to A, E, and F, as follows.

```

SUBROUTINE GCSE2
.
.
.
T1 = -(J * B + SQRT(C))
IF (K .LT. L) THEN
    A = (C * 4) / T1
ELSE
    E = (E * 4) / T1
ENDIF
F = (B * 4) / T1
.
.
.
END

```

The analogous C code follows.

```

void gcse2()
{
.
.
.
    if (k < l)
        a = (c * 4) / -(j * b + sqrt(c));
    else
        e = (e * 4) / -(j * b + sqrt(c));
        f = (b * 4) / -(j * b + sqrt(c));
.
.
.
}

```

As with the Fortran example, the compiler recognizes that the common subexpression is used before and after the `if` statement. It saves the value of the subexpression in the temporary variable `t1` before the `if` statement and uses this variable to compute the values of `a`, `e`, and `f`, as shown in the following example.

```

void gcse2()
{
...
    t1 = -(j * b + sqrt(c));
    if (k < l)
        a = (c * 4) / t1;
    else
        e = (e * 4) / t1;
        f = (b * 4) / t1;
...
}

```

---

## Code motion

Code motion is the movement of invariant expressions out of loops. An invariant expression yields the same result on every iteration of a loop.

In the following Fortran example, all variables used in the assignment to A remain invariant within the loop.

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
DO I = 1, 10
  A = C / (-(E * B) + SQRT(C))
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

The optimized code follows.

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
A = C / (-(E * B) + SQRT(C))
DO I = 1, 10
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

In C:

```
void gcm() {
    float ar[10]
    .
    .
    .
    for(i=0;i<10;i++) {
        a = c/(-(e*b) + sqrt(c));
        ar[i] = a+b+c;
    }
    .
    .
    .
}
```

After optimization:

```
void gcm() {
    float ar[10]
    .
    .
    .
    a = c/(-(e*b) + sqrt(c));
    for(i=0;i<10;i++)
        ar[i] = a+b+c;
    .
    .
    .
}
```

If an invariant expression does not lie on a path to all loop exits, the compiler does not move the invariant expression unless you use the `-uo` (potentially unsafe optimizations) compiler option.

For more information about using the `-uo` option, refer to Chapter 9, "Potentially unsafe optimizations."

---

## Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation (possibly nonarithmetic) that executes more quickly. Such replacements are called strength reductions.

### Explicit arithmetic reductions

The compiler can reduce the strength of various arithmetic operations. For example, the compiler transforms integer

multiplication on positive numbers by 2, 4, 8, and 16 into integer shifts, as shown in the following Fortran code:

```
J * 2 becomes IISHFT(J, 1)
```

```
J * 4 becomes IISHFT(J, 2)
```

Multiplication involving integer constants is reduced to addition:

```
X * 2 becomes X + X
```

When the `-uo` (potentially unsafe optimizations) command line option is specified, division by a constant is reduced to multiplication, which is substantially faster:

```
X/C becomes D*X where D=1/C
```

Because `C` is a constant, `D` also is a constant, which can be computed at compile time.

### Induction variables and constants

The compiler can reduce the strength of operations to optimize loop induction variables and loop constants. Multiplications within a loop that calculate the address of a subscripted variable are often candidates for strength reduction.

The compiler does not reduce operations that only involve `REAL`, `float` or `double` variables. Because floating-point arithmetic is imprecise, reduced operations do not always yield equivalent results. If an expression does not lie on a path to all loop exits, the compiler does not reduce the expression unless you use the `-uo` option.

In the following Fortran example, the compiler recognizes that `I` is incremented by 2 on each iteration and that `X` is incremented by `2*C`, a loop constant.

```
      SUBROUTINE GSR
      I = 1      !induction var
10     X = I * C !loop induction value
      .
      .
      .
      I = I + 2
      IF(I .LE. 100) GOTO 10
      .
      .
      .
      END
```

The analogous C code follows:

```
void gsr()
{
    int i = 1;      /* induction var */
    ...
    do {
        x = i * c; /* loop induction value */
        ...
        i += 2;
    } while(i<=100);
}
```

As shown in the following optimized version, the compiler produces code that calculates  $2 * C$  only once and increments  $X$  by the value saved in  $T2$  instead of calculating  $I * C$  on every iteration.

```
        SUBROUTINE GSR
        I = 1
        T1 = C
        T2 = 2 * C
10      X = T1
        .
        .
        .
        T1 = T1 + T2
        I = I + 2
        IF(I .LE. 100) GOTO 10
        .
        .
        .
        END
```

In C:

```
void gsr()
{
    int i = 1;
    ...
    t1 = c;
    t2 = c + c;
    do {
        x = t1;
        ...
        t1 += t2;
        i += 2;
    } while(i <= 100);
}
```

---

## Global register allocation

Scalar variables can often be stored in registers, eliminating the need for costly main memory accesses. Global register allocation (GRA) attempts to store commonly-referenced scalar variables in registers throughout the code in which they are most frequently accessed. Consider the following Fortran example:

```
DO I = 1, N
  A(I) = X
  .
  .
  .
ENDDO
```

Here, X is referenced on every iteration of the loop. Eliminating loads and stores of X to main memory can substantially improve performance. Using GRA, the compiler generates code equivalent to that shown below:

```
REG = X
DO I = 1, N
  A(I) = REG
  .
  .
  .
ENDDO
X = REG
```

Where REG represents a register.

The analogous C example follows:

```
for(i=0;i<n;i++) {
  a[i] = x;
  .
  .
  .
}
```

After GRA:

```
REG = x;
for(i=0;i<n;i++){
  a[i] = REG;
  .
  .
  .
}
x = REG;
```

The compiler automatically determines which scalar variables are the best candidates for GRA and allocates registers accordingly.

GRA can sometimes cause wrong answers in Fortran code that violates ANSI standard argument-passing conventions. The problem arises when a constant is passed into a subroutine which potentially attempts to assign to it. If such an assignment executes, a bus error will occur; however, some codes may conditionally execute the assignment based on whether the argument is a variable or constant in the calling program.

In such cases, GRA may be unaware that the subroutine's dummy arguments may represent constant actual arguments. If the dummy arguments are allocated registers, a bus error will result even if the executable never attempts to assign to the constant, because GRA will always generate code to store the register back to the dummy argument variable at some point.

Consider the following Fortran example:

```
.
.
.
CALL ASSIGNER(1,IMAX,A)
CALL ASSIGNER(0,10,A)
.
.
.
SUBROUTINE ASSIGNER(IVAR,I,A)
INTEGER A(100,100)
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    . ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
RETURN
END

INTEGER FUNCTION TEST(ITEST)
.
.
.
```

Here, the IVAR argument indicates whether the I dummy argument is a variable or constant. If it is a variable, it is assigned in the IF statement.

GRA, unaware that I may be a constant, will allocate a register for it (assuming nothing in the missing code prevents it from doing so). When the register is stored back to I at runtime, a runtime error results.

The following Fortran example shows code that is equivalent to code produced by the GRA optimization for the subroutine ASSIGNER.

```
SUBROUTINE ASSIGNER(IVAR, I, A)
INTEGER A(100,100)
REG = I          ! I PLACED IN REGISTER
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    . ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
I = REG ! REG STORED BACK TO I--IMPOSSIBLE
RETURN ! IF I IS A CONSTANT
END
```

This situation can be avoided by specifying the `-nga` compiler option. `-nga` disables global register allocation for arguments passed by reference. If the above example is compiled with `-nga`, I will never be allocated a register, and the code will work as expected.

GRA can also sometimes cause problems when parallel threads attempt to update a shared variable that has been allocated a register. In this case, each parallel thread will allocate a register for the shared variable; it is then unlikely that the copy in main memory will be updated correctly as each thread executes.

Parallel assignments to the same shared variables from multiple threads only make sense if the assignments are contained inside critical or ordered sections, or are executed conditionally based on thread ID. GRA will not allocate registers for shared variables that are assigned within critical or ordered sections, as long as the sections are implemented using compiler directives or

`sync_routine`-defined functions (refer to Chapter 6, “Advanced shared memory programming”). However, for conditional assignments based on thread ID, GRA may allocate registers which may cause wrong answers when stored.

In such cases, GRA can be disabled only for shared variables which are visible to multiple threads by specifying the `-ngs` compiler option.

In procedures with large numbers of loops, GRA can contribute to long compile times; therefore, GRA is only performed if the number of loops in the procedure is below a pre-determined limit. You can remove this limit (and possibly significantly increase compile time and compile-time memory usage) by specifying the `-mr1` compiler option.

GRA is enabled by default at optimization levels `-O1` and higher.

---

## **-O2 Level optimizations**

The primary goal of `-O2` optimizations is *data localization*. Data localization keeps heavily used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

Loops that manipulate arrays are the main candidates for localization optimizations. Most of these loops are eligible for the various transformations the compiler performs at level `-O2` to achieve localization. These transformations are explained in detail in this section.

Many loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, the total number of loops replicated by all transformations is limited by default. Each optimization is allocated a percentage of this limit; if an optimization hits its limit, an advisory is issued and the optimization is disabled for any following loops. You can increase this limit (and possibly increase your program’s compile time and compile-time memory usage) by specifying the `-mr1` compiler option.

In addition to localization, the compiler performs global instruction scheduling and all basic block and machine-instruction level optimizations at optimization level `-O2`.

## **Note**

Most of the subsections that follow present code examples that demonstrate the optimization in question by showing the original code first and optimized code second. While the optimized code is shown in the same language as the original code, this is for illustrative purposes only. The compiler actually generates an intermediate language after performing these optimizations.

---

## Why localize?

The benefits of localization are best illustrated through an example.

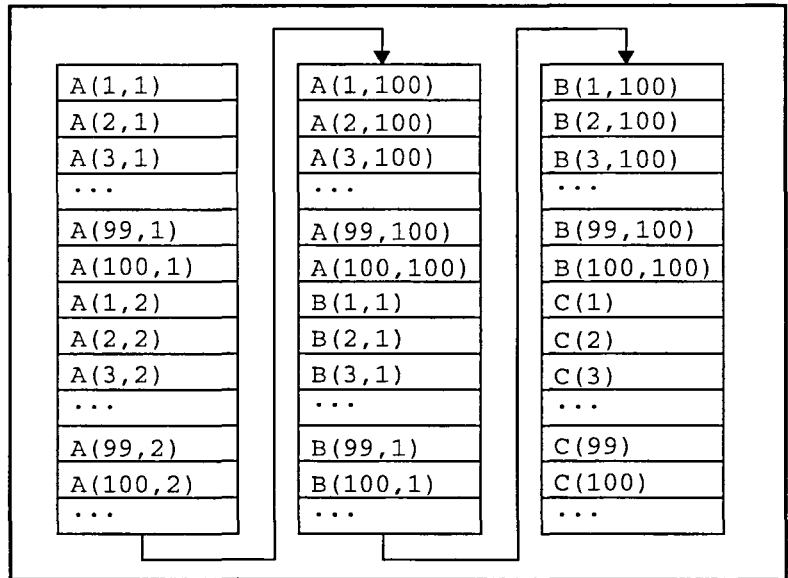
Consider the following Fortran code:

```
REAL*4 A(100, 100), B(100, 100), C(100)
COMMON /BLK1/ A, B, C
...
DO J = 1, 100
  DO I = 1, 100
    A(I, J) = B(I, J) * C(I)
  ENDDO
ENDDO
```

The compiler will recognize that this loop nest is too small to benefit from blocking. As the loop is written, all three arrays can be reused.

First, notice that this loop operates on three REAL\*4 arrays, which total less than 80 kbyte in size. All the elements of these arrays will fit easily into the processor data cache, so elements being overwritten will not be a problem. The arrays are stored in COMMON to eliminate the possibility of cache thrashing, as described in Chapter 2, "Architecture overview."

Now let's examine how these arrays are stored in memory. Fortran arrays are stored in column-major order, meaning that the elements of the first column are stored contiguously, followed by the elements of the second column and so on, as shown in Figure 11. Note that contiguous storage of the arrays in this illustration is due to the COMMON block storage of the arrays.



**Figure 11** Array storage in memory

The boxes in Figure 11 represent contiguous virtual addresses; these could map to physical addresses in the CTIcache or in main memory. In any case, assuming the arrays have not been accessed before we reach the  $J$  loop and therefore are not in the processor data cache, they must be encached there as the loop executes.

On the first iteration of the loop,  $A(1, 1)$ ,  $B(1, 1)$  and  $C(1)$  will be fetched from memory as part of separate 32-byte cache lines. These elements will be positioned at random relative to the cache line boundaries; in other words, there is no way of knowing whether  $A(1, 1)$  is the first element in its cache line, or if it is positioned elsewhere. While it is likely that each cache line will contain some reusable elements of the array it is fetching from, this is not guaranteed on the first fetch. However, when any present reusable elements are used and the second fetch from memory takes place, the cache line in question will begin with the element being fetched and also contain the 7 following elements. Thus, after the first element is fetched for each array, fetching  $A(I, 1)$  will encache  $A(I:I+7, 1)$ ; fetching  $B(I, 1)$  will encache  $B(I:I+7, 1)$  and fetching  $C(I)$  will encache  $C(I:I+7)$ . The  $I$  loop will then iterate 7 more times without needing to look beyond the processor data cache for data, boosting performance significantly.

Another significant performance boost happens when the  $I$  loop finishes executing for  $J = 1$ . At this point, all of  $C$  has been encached. While each successive column of  $A$  and  $B$  will have to be fetched a cache line at a time from memory,  $C$  will always be

available immediately from the processor data cache, saving, in this simple example, 13 out-of-cache memory accesses.

More complicated loops, such as the matrix multiply algorithm, and loops that manipulate much larger arrays can often be transformed by the compiler such that far more data reuse is possible, resulting in greater performance.

The following sections explain the loop transformations that aid data localization.

---

## Strip mining

Strip mining is a fundamental -O2 transformation that is used by loop blocking and, in a sense, by parallelization.

Strip mining involves splitting a single loop into a nested loop; the resulting inner loop iterates over a section or *strip* of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, i.e. to achieve the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's *strip length*.

Consider the following Fortran code.

```
DO I = 1, 10000
  A(I) = A(I) * B(I)
ENDDO
```

Strip mining this loop using a strip length of 1000 yields the following loop nest:

```
DO IO OUTER = 1, 10000, 1000
  DO ISTRIP = IO OUTER, IO OUTER+999
    A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
  ENDDO
ENDDO
```

In this loop, the strip length integrally divides the number of iterations, so the loop is evenly split up. If the iteration count was not an integral multiple of the strip length, e.g. if I went from 1 to 10500 rather than 1 to 10000, the final iteration of the strip loop would execute 500 iterations instead of 1000.

An analogous C example follows:

```
for(i=0;i<10000;i++)
  a[i] = a[i] * b[i];
```

After strip mining with a strip length of 1000

```
for(iout=0;iout<10000;iout+=1000)
  for(istrip=iout;istrip<iout+1000;istrip++)
    a[istrip] = a[istrip] * b[istrip];
```

In and of itself, strip mining is not profitable. However, strip mining is essential to the highly profitable loop blocking optimization, which is described in a following section of this chapter.

---

## Loop distribution

Loop distribution is another fundamental -O2 transformation that is necessary for some more advanced transformations. These advanced transformations require that all calculations in a nested loop be performed inside the innermost loop. To facilitate this, loop distribution transforms complicated nested loops into several simple loops (or nests) that contain all computations inside the body of the innermost loop.

Consider the following Fortran code:

```
DO I = 1, N
  B(I, 1) = 0
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```

Loop distribution creates three copies of the I loop, separating the nested J loop from the assignments to arrays B and D. In this way, all three assignments are moved to innermost loops, as shown in the following transformed code.

```
DO I = 1, N
  B(I, 1) = 0
ENDDO
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
ENDDO
DO I = 1, N
  D(I) = E(I) + A(I)
ENDDO
```

An analogous C example follows:

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

This loop is distributed as shown below.

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

Distribution can improve efficiency by reducing the number of memory references per loop iteration, and can reduce cache thrashing. It also creates more opportunities for interchange.

Loop distribution can be disabled for specific loops by specifying the `no_distribute` directive or `pragma` immediately before the loop. In Fortran, it has the following form:

```
C$DIR NO_DISTRIBUTE
```

In C:

```
#pragma _CNX no_distribute
```

---

## Loop interchange

The compiler interchanges nested loops for the following reasons:

- To facilitate other transformations.
- To relocate the loop that is the most profitable to parallelize so that it is outermost (at optimization level `-O3` only).
- To optimize inner-loop memory accesses.

Consider the Fortran matrix addition algorithm that follows.

```
DO I = 1, N
    DO J = 1, M
        A(I, J) = B(I, J) + C(I, J)
    ENDDO
ENDDO
```

This loop accesses the arrays A, B and C row by row, which, in Fortran, is very inefficient. Interchanging the I and J loops, as

shown in the following example, will facilitate column by column access.

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

Unlike Fortran, C accesses arrays in row-major order. An analogous example in C, then, employs an opposite nest ordering, as shown below.

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

Interchange facilitates row-by-row access. The interchanged loop is shown below.

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

---

## Loop blocking

Loop blocking is a combination of strip mining and interchange that maximizes data localization. It is provided primarily to deal with nested loops that manipulate arrays that are too large to fit into the cache. Under certain circumstances, loop blocking allows reuse of these arrays by transforming the loops that manipulate them so that they manipulate strips of the arrays that fit into the cache. Effectively, a blocked loop accesses array elements in sections that are sized to optimally fit in the cache.

### Data reuse

Data reuse is important to understand when discussing blocking. There are two types of data reuse associated with loop blocking: *spatial* reuse and *temporal* reuse.

Spatial reuse is using data which was encached as a result of fetching another piece of data from memory. Remember that on an SPP Series system, data is fetched by cache lines; 32 bytes of data is encached on every fetch. On the initial fetch of array data from memory within a stride-one loop, the requested item can be located anywhere in the 32 bytes, unless the array is properly aligned (refer to the "Data alignment" section of Chapter 2, "Architecture overview"). Starting with the second memory fetch (which will not happen until any usable elements obtained on the

first fetch are used), the requested data is at the beginning of the cache line, and the rest of the cache line will contain subsequent array elements. For a REAL\*4 array, this means the requested element and the 7 following elements are encached on each fetch after the first. If any of these 7 elements could then be used, say on any subsequent iterations of the loop, the loop would be exploiting spatial reuse. For loops with strides greater than one, spatial reuse can still occur; however, the cache lines may contain fewer usable elements.

Temporal reuse is using the same data item on more than one iteration of the loop. An array element whose subscript does not change as a function of the iterations of a surrounding loop exhibits temporal reuse in the context of the loop.

Loops containing either temporal or spatial reuse are candidates for blocking. Blocking exploits spatial reuse by ensuring that once fetched, cache lines are not overwritten until their spatial reuse is exhausted. Temporal reuse is similarly exploited.

### Reuse example

The following Fortran loop contains arrays that are candidates for both spatial and temporal reuse.

```
REAL*4 A(100,100), B(100,100), C(100)
COMMON /BLK1/ A, B, C
.
.
.
DO J = 1, 100
  DO I= 1, 100
    A(I,J) = B(J,I) + C(I)
  ENDDO
ENDDO
```

As written, this loop gives spatial reuse on the A and B arrays, and both spatial and temporal reuse on the C array. Because the arrays are in a COMMON block and each array is of a total length that is integrally divisible by the CTIcache line length, we know that each array will begin on a CTIcache line boundary. All cache lines fetched will be full of reusable data. Spatial reuse is achieved on the A array because every 8th iteration of the I loop fetches a cache line containing 8 of its elements; the 7 iterations between main memory accesses can proceed with virtually no load delays. This continues throughout the entire range of the J loop.

Similar spatial reuse is achieved on the B array. During the first iteration of J, every referenced element of B, along with its containing cache line, will be fetched from memory. All the elements contained in this cache line will be reusable on some

subsequent iteration of one of the loops. On subsequent iterations of  $J$ , a cacheline will be fetched from memory only if the required element was not previously encached, and all the elements it contains will be usable. However, keep in mind that fetches are a function of  $I$  and may occur for different  $J$  values (after  $J = 1$ ) based on the value of  $I$ . Since  $B$ 's row index is  $J$ , any unused encached elements are used on the subsequent iterations of  $J$  for a given value of  $I$ . Though the data is not used as immediately as it is for  $A$ , spatial reuse is still fully exploited.

Spatial reuse is similarly achieved on the  $C$  array, but only for  $J = 1$ . Assuming the loop is compiled as written, when the first iteration of  $J$  finishes,  $C$  is completely contained in the processor data cache and will remain there for the duration of  $J$ .  $C$  can then be temporally reused for every subsequent iteration of  $J$ .

Note that this loop does not require blocking to achieve this reuse because the arrays only occupy a total of less than 80 kbytes, so they fit easily into the cache. Spatial reuse is graphically illustrated for a similar but more realistic example in the following section.

### Blocking example—simple loop

In order to achieve reuse in more realistic examples which manipulate arrays that won't all fit in the cache, blocking strip mines the inner loop. The new innermost loop has an iteration space selected so that all the array elements it references will fit into the cache without overwriting themselves. The new loop nest is then interchanged such that the innermost loop is "blocked" from overwriting its strips.

Consider the following Fortran example:

```
REAL*8 A(1000,1000),B(1000,1000)
REAL*8 C(1000),D(1000)
COMMON /BLK2/ A, B, C
.
.
.
DO J = 1, 1000
  DO I = 1, 1000
    A(I,J) = B(J,I) + C(I) + D(J)
  ENDDO
ENDDO
```

Here the array elements occupy nearly 16 Mbytes of memory, and blocking becomes quite profitable.

First the compiler strip mines the I loop:

```
DO J = 1, 1000
  DO IOUIT = 1, 1000, IBLOCK
    DO I = IOUIT, IOUIT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

IBLOCK is the block factor (it's also the strip mine length) the compiler chooses based on the size of the arrays and size of the cache. Note that this example assumes the chosen IBLOCK divides 1000 evenly.

Now the compiler interchanges the J and IOUIT loops to block the I loop.

```
DO IOUIT = 1, 1000, IBLOCK
  DO J = 1, 1000
    DO I = IOUIT, IOUIT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

This new nest now accesses IBLOCK rows of A and IBLOCK columns of B for every iteration of J. At every iteration of IOUIT, the nest accesses 1000 IBLOCK-length columns of A, or an IBLOCK × 1000 chunk of A, and 1000 IBLOCK-width rows of B are accessed. This is illustrated in Figure 12.

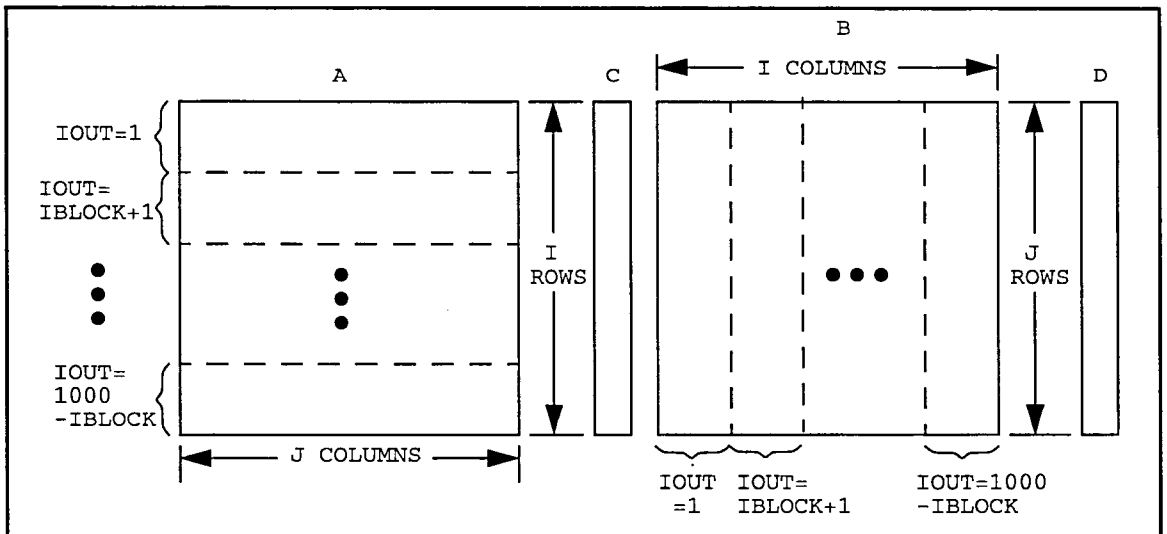


Figure 12 Blocked array access

Fetches of A encache the needed element and the 3 elements that are used in the 3 subsequent iterations, giving spatial reuse on A. Since the I loop traverses columns of B, fetches of B encache extra elements that will not be spatially reused until J increments. IBLOCK is chosen by the compiler to efficiently exploit spatial reuse of both A and B. Figure 13 illustrates how cache lines of each array are fetched (A and B both start on cache line boundaries because they are in COMMON and of lengths integrally divisible by the CTIcache line length). The shaded area represents the initial cache line fetched.

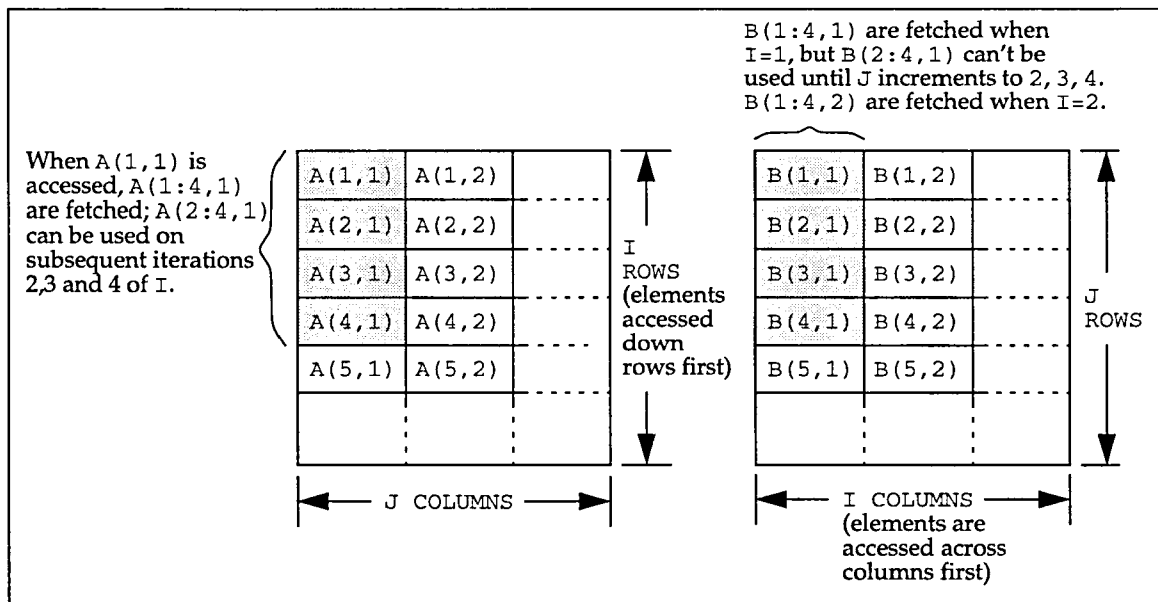


Figure 13 Spatial reuse of A and B

Typically, IBLOCK elements of C will remain in the cache for several iterations of J before being overwritten, giving temporal reuse on C for those iterations. By the time any of the arrays are overwritten, all spatial reuse has been exhausted. The load of D is hoisted out of the I loop so that it remains in a register for all iterations of I.

## Blocking example—matrix multiply

The more complicated matrix multiply algorithm, which follows, is a prime candidate for blocking.

```
REAL*8 A(1000,1000),B(1000,1000),C(1000,1000)
COMMON /BLK3/ A, B, C
.
.
.
DO I = 1,1000
  DO J = 1, 1000
    DO K = 1,1000
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

This loop is blocked as shown below.

```
DO IOUT = 1, 1000, IBLOCK
  DO KOUT = 1, 1000, KBLOCK
    DO J = 1, 1000
      DO I = IOUT, IOUT+IBLOCK-1
        DO K = KOUT, KOUT+KBLOCK-1
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here, we get spatial reuse of B with respect to the K loop; temporal reuse of B with respect to the I loop; spatial reuse of A with respect to the I loop; temporal reuse of A with respect to the J loop; spatial reuse of C with respect to the I loop; and temporal reuse of C with respect to the K loop.

An analogous C example follows:

```
static double a[1000][1000], b[1000][1000];
static double c[1000][1000];
.
.
.
for(i=0;i<1000;i++)
  for(j=0;j<1000;j++)
    for(k=0;k<1000;k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Convex C interchanges and blocks the loop in this example to provide optimal access efficiency for the row-major C arrays. The blocked loop is shown below.

```
for (jout=0; jout<1000; jout+=jblk)
  for (kout=0; kout<1000; kout+=kblk)
    for (i=0; i<1000; i++)
      for (j=jout; j<jout+jblk; j++)
        for (k=kout; k<kout+kblk; k++)
          c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

As you can see, the interchange was done differently because of C's different array storage. This code yields spatial reuse of *b* with respect to the *j* loop; spatial reuse of *a* with respect to the *k* loop; temporal reuse of *a* with respect to the *j* loop; spatial reuse on *c* with respect to the *j* loop; and temporal reuse on *c* with respect to the *k* loop.

Blocking is inhibited when loop interchange is inhibited. If a candidate loop nest contains loops that cannot be interchanged, blocking will not be performed.

### Blocking directives, pragmas and options

Loop blocking can be disabled for specific loops using the `NO_BLOCK_LOOP` compiler directive and pragma. You can advise the compiler to use a specific block factor via the `BLOCK_LOOP` directive and pragma. In Fortran, these directives have the following form:

```
C$DIR NO_BLOCK_LOOP
C$DIR BLOCK_LOOP [ (BLOCK_FACTOR = n) ]
```

In C, these directives have the following form:

```
#pragma _CNX block_loop[ (block_factor = n) ]
#pragma _CNX no_block_loop
```

In the `BLOCK_LOOP` directive and pragma, *n* is the requested block factor, which must be an integer constant. The compiler will use this value as stated, so for best performance, the block factor multiplied by the data type size of the data in the loop should be an integral multiple of the cache line size. In absence of the `block_factor` argument, this directive is useful for indicating which loop in a nest to block. In this case, the compiler will pick an appropriate block factor.

These directives affect the loop that immediately follows them.

Reconsider the matrix multiply example, this time with a `BLOCK_LOOP` directive.

```
REAL*8 A(1000,1000),B(1000,1000),C(1000,1000)
COMMON /BLK3/ A, B, C
.
.
.
      DO I = 1,1000
        DO J = 1, 1000
C$DIR      BLOCK_LOOP(BLOCK_FACTOR = 112)
          DO K = 1,1000
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO
```

We know from the original example involving this code that the compiler blocks the I and K loops. In this example, the `BLOCK_LOOP` directive instructs the compiler to use a block factor of 112 for the K loop. This is an efficient blocking factor for this example, because  $112 \times 8$  bytes = 896 bytes, and  $896/32$  bytes (the cache line size) = 28, which is an integer, so partial cache lines will not be needed. The compiler-chosen value is still used on the I loop.

To disable blocking for all loops, use the `-noblock` compiler option. To specify a blocking factor for all loops, use the `-blockloop n` compiler option, where *n* is the blocking factor to be used for all loops that the compiler blocks.

As with all optimizations that replicate code, the number of new loops created when the compiler strip mines for blocking is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler will issue an advisory and blocking will be disabled for all following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mr1` compiler option.

---

## Loop fusion

Loop fusion involves creating one loop out of two or more neighboring loops that have identical loop bounds and trip counts. This reduces loop overhead, memory accesses, and register usage, and can lead to other optimizations such as assignment substitution, redundant-assignment elimination and loop blocking. By potentially reducing the number of parallelizable loops in a program, loop fusion can greatly reduce

parallelization overhead, since fewer spawns and joins will be necessary.

Consider the following Fortran code:

```
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
DO J = 1, N
  IF(A(J) .LT. 0) A(J) = B(J)*B(J)
ENDDO
```

These two loops can be fused into the following loop:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  IF(A(I) .LT. 0) A(I) = B(I)*B(I)
ENDDO
```

And this loop can be further optimized by applying assignment substitution and common-subexpression elimination as shown in the following code:

```
DO I = 1, N
  TEMP1 = B(I)
  TEMP2 = TEMP1 + C(I)
  IF(TEMP2 .LT. 0) TEMP2 = TEMP1*TEMP1
  A(I) = TEMP2
ENDDO
```

Here assignment substitution is exploited when the array reference to  $B(I)$  is replaced by the scalar  $TEMP1$ , and the reference to  $A(I)$  is replaced by  $TEMP2$ .  $TEMP2$  is used to hold the value of  $A(I)$  until there's no chance the value will change, thus eliminating redundant assignments to  $A(I)$ .

An analogous C example follows:

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

After loop fusion, assignment substitution and common-subexpression elimination, the compiler produces code equivalent to the following:

```
for(i=0; i<n; i++) {
  temp1 = b[i];
  temp2 = temp1 + c[i];
  if(temp2 < 0) temp2 = temp1*temp1;
  a[i] = temp2;
}
```

Occasionally loops that do not appear to be fusible become fusible as a result of compiler transformations that precede fusion. For instance, peeling or interchanging a loop may make it suitable for fusing with another loop.

Loop fusion is especially beneficial when applied to Fortran 90 array assignments. The compiler translates these statements into loops; when such loops do not contain code that would inhibit fusion, they can be fused.

Consider the following Fortran example:

```
REAL A(1000,1000), B(1000,1000), C(1000,1000)
.
.
.
C = 2.0 * B
A = A + B
```

The compiler would first transform these Fortran 90 array assignments into loops, generating code similar to that shown below.

```
DO TEMP1 = 1, 1000
  DO TEMP2 = 1, 1000
    C(TEMP1, TEMP2) = 2.0 * B(TEMP1, TEMP2)
  ENDDO
ENDDO
DO TEMP3 = 1, 1000
  DO TEMP4 = 1, 1000
    A(TEMP3, TEMP4) = A(TEMP3, TEMP4) + B(TEMP3, TEMP4)
  ENDDO
ENDDO
```

These two loops would then be fused as shown in the following loop nest:

```
DO TEMP1 = 1, 1000
  DO TEMP2 = 1, 1000
    C(TEMP1, TEMP2) = 2.0 * B(TEMP1, TEMP2)
    A(TEMP1, TEMP2) = A(TEMP1, TEMP2) + B(TEMP1, TEMP2)
  ENDDO
ENDDO
```

And the compiler can apply further optimizations to this new nest.

Loop fusion is enabled by default and by the `-f1` compiler option. It can be disabled by specifying the `-nf1` compiler option. The `no_fuse` compiler directive and pragma can be used to disable loop fusion for specific loops. In Fortran, it has the following form:

```
C$DIR NO_FUSE
```

In C, it has the following form:

```
#pragma _CNX no_fuse
```

`no_fuse` is effective for the immediately following loop only.

When fusion is disabled for the program, you can request it for particular loops using the `prefer_fuse` directive and `pragma`. In Fortran, it has the following form:

```
C$DIR PREFER_FUSE
```

In C, it has the following form:

```
#pragma _CNX prefer_fuse
```

A `prefer_fuse` directive or `pragma` must immediately precede the loop for which you want to enable fusion. If fusion is enabled for the program, `prefer_fuse` has no affect. If fusion is disabled for the program, `prefer_fuse` must be specified on two or more neighboring loops, and will override the `-nfl` option.

## Loop unrolling

Loop unrolling involves increasing a loop's step value and replicating the loop body, with each replication appropriately offset from the induction variable so that all iterations are performed given the new step.

Unrolling can be total or partial. Total unrolling involves eliminating the loop structure completely, replicating the loop body a number of times equal to the iteration count, and replacing the iteration variable with constants. This only makes sense for loops with small iteration counts. Consider the following Fortran example.

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

This loop is completely unrolled as shown in the following example:

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
A(4) = B(4) + C(4)
```

Partial unrolling is performed on loops with larger or unknown iteration counts. It retains the loop structure, but replicates the body a number of times equal to the *unrolling depth* (also known as the *unroll factor*) and adjusts references to the iteration variable accordingly.

Consider the following Fortran example.

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

This example can be unrolled to a depth of 4 as shown below.

```
DO I = 1, 100, 4
  A(I) = B(I) + C(I)
  A(I+1) = B(I+1) + C(I+1)
  A(I+2) = B(I+2) + C(I+2)
  A(I+3) = B(I+3) + C(I+3)
ENDDO
```

Each iteration of the loop now computes 4 values of A instead of 1 value.

An analogous C example follows:

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

This can be unrolled to a depth of 4 as shown in the following code:

```
for(i=0;i<100;i+=4) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

Loop unrolling improves efficiency by eliminating loop overhead; in totally unrolled loops, the overhead is completely eliminated, and in partially unrolled loops the number of tests and induction variable increments are reduced. Unrolling can also create opportunities for other optimizations, such as improved register use and more efficient scheduling.

Convex compilers perform loop unrolling by default at optimization levels `-O2` and higher. The compiler completely unrolls loops with iteration counts determinable at compile time to be less than 5. Loops with undeterminable iteration counts or determinable counts of 5 or more are partially unrolled.

Unrolling is enabled by default and through use of the `-ur` compiler option. It can be disabled for all loops by specifying the `-nur` compiler option. You can specify an unroll factor by using the `-urn n` option, where `n` is the unroll factor for all unrolled loops.

When `-nur` is used, unrolling can also be specified for individual loops using the `unroll` directive and `pragma`. In Fortran, this directive has the following form:

```
C$DIR UNROLL [ (UNROLL_FACTOR=n) ]
```

The C pragma has the following form:

```
#pragma _CNX unroll [ (unroll_factor=n) ]
```

Where the optional `unroll_factor=n` argument allows you to specify an unroll factor of *n* for the loop in question.

When unrolling is enabled for the entire program, it can be disabled for individual loops using the `no_unroll` directive and `pragma`. In Fortran, this directive has the following form:

```
C$DIR NO_UNROLL
```

The C pragma has the following form:

```
#pragma _CNX no_unroll
```

## Note

**Unrolling is only performed on innermost loops. If you use `unroll` or `no_unroll` on a loop nest, you must specify it on the loop which ends up, after any interchanges performed by the compiler, to be innermost.**

To ensure that the directive or pragma is on the innermost loop, you should compile the nest in question and determine from the optimization report (which is discussed in Appendix C) which loop is innermost after compilation, then place the directive or pragma on that loop in your source. Placing the directive or pragma on a loop that is not innermost after compilation will have no effect.

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll optimization is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler will issue an advisory and unrolling will be disabled for all following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mr1` compiler option.

---

### Loop unroll and jam

The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory

accesses and sometimes allows better exploitation of certain machine instructions.

Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that can be temporally reused with respect to some loop other than the innermost.

Blocked loops are often prime candidates for unroll and jam because their deep nesting provides many candidates for temporal reuse with respect to the non-innermost loops.

Consider our blocked matrix multiply loop:

```
DO IOUT = 1, N, ISTRIP
  DO KOUT = 1, N, KSTRIP
    DO J = 1, N
      DO I = IOUT, IOUT+ISTRIP-1
        DO K = KOUT, KOUT+KSTRIP-1,
          C(I,J) = C(I,J) + A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here, the compiler can exploit a maximum of 3 registers: one for  $C(I, J)$ , one for  $A(I, K)$ , and one for  $B(K, J)$ .

Register exploitation can be vastly increased on this loop by unrolling and jamming the  $J$  and  $I$  loops. First, the compiler unrolls these loops. To simplify the illustration, we will use an unrolling factor of 2 for both  $I$  and  $J$ . This is the number of times the contents of the loops will be replicated.

The following Fortran example shows this replication:

```

DO IOUT = 1, N, ISTRIP
DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
        C(I,J) = C(I,J)+A(I,K)*B(K,J)
      ENDDO
      DO K = KOUT, KOUT+KSTRIP-1
        C(I+1,J) = C(I+1,J)+A(I+1,K)*B(K,J)
      ENDDO
    ENDDO
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
        C(I,J+1) = C(I,J+1)+A(I,K)*B(K,J+1)
      ENDDO
      DO K = KOUT, KOUT+KSTRIP-1
        C(I+1,J+1)=C(I+1,J+1)+A(I+1,K)*B(K,J+1)
      ENDDO
    ENDDO
  ENDDO
ENDDO
ENDDO
ENDDO

```

The "jam" part of unroll and jam occurs when the loops are fused back together, to create the following:

```

DO IOUT = 1, N, ISTRIP
DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
        C(I,J) = C(I,J)+A(I,K)*B(K,J)
        C(I+1,J) = C(I+1,J)+A(I+1,K)*B(K,J)
        C(I,J+1) = C(I,J+1)+A(I,K)*B(K,J+1)
        C(I+1,J+1)= C(I+1,J+1)+A(I+1,K)*B(K,J+1)
      ENDDO
    ENDDO
  ENDDO
ENDDO
ENDDO

```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original blocked loop could use no more than 3 registers. This unrolled-and-jammed loop can use 8, one for each of the following references:  $C(I, J)$ ,  $A(I, K)$ ,  $B(K, J)$ ,  $C(I+1, J)$ ,  $A(I+1, K)$ ,  $C(I, J+1)$ ,  $B(K, J+1)$ ,  $C(I+1, J+1)$ . Fewer loads and stores per operation are required because all of the registers containing these elements

are referenced at least twice. This particular example can also benefit from the PA-RISC `FMPYADD` instruction, which doubles the speed of the operations in the body of the loop by simultaneously performing unrelated adds and multiplies.

Remember, this is a very simplified example. In reality, the compiler attempts to exploit as many of the PA-RISC processor's registers as possible. For the matrix multiply algorithm used here, the compiler would pick a larger unrolling factor, creating a much larger  $K$  loop body. This would result in increased register exploitation and fewer loads and stores per operation.

Unroll and jam is enabled by default and through use of the `-uj` option at optimization levels `-O2` and higher. You can disable unroll and jam by specifying the `-nuj` option. You can specify an unrolling factor by using the `-ujn n` option, where  $n$  is the desired unrolling factor for all loops which the compiler unrolls and jams.

You can specify unroll and jam for individual loops by using the `unroll_and_jam` compiler directive and pragma. In Fortran, it has the following form:

```
C$DIR UNROLL_AND_JAM[ (UNROLL_FACTOR= $n$ ) ]
```

The C pragma has the following form:

```
#pragma _CNX unroll_and_jam[ (unroll_factor= $n$ ) ]
```

Where the optional `unroll_factor= $n$`  argument allows you to specify an unroll factor for the loop in question.

You can disable unroll and jam for individual loops using the `no_unroll_and_jam` directive and pragma. In Fortran, it has the following form:

```
C$DIR NO_UNROLL_AND_JAM
```

The C pragma has the following form:

```
#pragma _CNX no_unroll_and_jam
```

## Note

**All of these directives and pragmas apply to the immediately following loop nest. Since unroll and jam is only performed on nested loops, you must ensure that the directive or pragma is specified on a loop that, after any compiler-initiated interchanges, ends up being non-innermost. You can determine which loop in a nest will be innermost by compiling the nest without any directives and examining the optimization report.**

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll and jam optimization is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler will issue an advisory and unroll and jam will be disabled for all

following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mr1` compiler option.

---

## IF-DO and if-for optimizations

These optimizations modify loops containing tests to improve performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of branches that must be executed, thereby improving performance.

In Fortran, these optimizations are performed on `DO` loops containing `IF` statements; in C, they are performed on `for` loops containing `if` statements. These optimizations fall into one of three categories: redundant test elimination, loop peeling, and test promotion. Each of these is described in detail below.

### Redundant-test elimination

Redundant-test elimination is the simplest of these optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code.

This optimization is especially relevant when you are optimizing FORTRAN 66 programs that contain `DO` loops surrounded by `IF` tests, as shown in the following example:

```
DO I = 1, N
  IF (I .GT. 0) THEN
    DO J = 1, I
      A(I,J) = 0
    ENDDO
  ENDIF
ENDDO
```

With the test removed, the loop looks like this:

```
DO I = 1, N
  DO J = 1, I
    A(I,J) = 0
  ENDDO
ENDDO
```

Here the explicit test `IF (I .GT. 0)` is redundant, since the test is implicit in the `DO` loop. It is therefore removed during redundant-test elimination.

Similarly, the test in the following C code always succeeds:

```
for (i=0; i<n; i++)
    if (i>-1)
        a[i] += b[i];
```

The `if` test is redundant because of the loop control, which initializes `i` to 0 and increments it by one each iteration. The compiler recognizes that the condition always occurs and removes the test:

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

Redundant-test elimination is always performed at optimization levels `-O2` and above.

### Loop boundary-value peeling

Loop boundary-value peeling involves removing the first iteration, last iteration, or first and last iterations of a loop to remove conditional tests from the loop. This is done when the loop contains a test or tests involving an explicit reference to the loop index variable that, when evaluated, either always causes or always prevents execution of the first iteration, last iteration, or both. Last-iteration peeling is also sometimes necessary when the compiler automatically privatizes a loop variable which is referenced after the loop. In this case, peeling is used to assign the last-iteration value to the private variable so that it can be later referenced.

Given the Fortran code shown below, the compiler automatically peels off the first and last tests and rewrites the loop to cover the remaining indexes.

```
DO I = 1, 100
    IF (I .EQ. 1) THEN
        A(I) = B(I)
    ELSE IF (I .EQ. 100) THEN
        A(I) = C(I)
    ELSE
        A(I) = -A(I)
    ENDIF
ENDDO
```

After peeling, the code looks like this:

```
A(1) = B(1)
DO I = 2, 99
    A(I) = -A(I)
ENDDO
A(100) = C(100)
```

The analogous C code follows:

```
for (i=0; i<100; i++)
  if (i==0)
    a[i]=b[i];
  else
    if (i==99)
      a[i]=c[i];
    else
      a[i]= -a[i];
```

After peeling:

```
a[0]= b[0];
for (i=1; i<99; i++)
  a[i]=-a[i];
a[99]=c[99];
```

In some cases, boundary-value peeling requires replicating large amounts of code, and this can greatly increase the size of the executable file. By default, the compiler peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or, if you wish to do so on a loop-by-loop basis, the `PEEL` Fortran compiler directive or `peel` C pragma.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option, the `PEEL_ALL` Fortran directive, or the `peel_all` C pragma. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

Using any of the compiler options, directives or pragmas described above *only* increases the loop replication limit for the loop peeling optimization; it does not increase the limit on the total number of replicated loops. This leaves a smaller percentage of the total limit for other loop-replicating options. To increase the total limit, specify the `-mr1` compiler option.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL` Fortran compiler directive or the `no_peel` C pragma. See Appendix A, "Compiler directives and pragmas," for more information.

## Note

Loop boundary-value peeling is not performed on loops that have no tests on boundary values. In other words, the compiler does not try to peel unpeelable loops.

## Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.

Consider the following Fortran loop.

```
DO I = 1, N
  IF (FOO .EQ. BAR) THEN
    A(I) = B(I)
  ELSE
    A(I) = 0
  ENDIF
ENDDO
```

Test promotion produces the following code.

```
IF (FOO .EQ. BAR) THEN
  DO I = 1, N
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1, N
    A(I) = 0
  ENDDO
ENDIF
```

In C:

```
for(i=0; i<n; i++)
  if (foo==bar)
    a[i]=b[i];
  else
    a[i]=0;
```

After test promotion:

```
if (foo==bar)
  for(i=0; i<n; i++)
    a[i]=b[i];
else
  for (i=0; i<n; i++)
    a[i]=0;
```

For loops containing large numbers of tests, loop replication can greatly increase the size of the code.

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler

promotes tests and replicates code up to a predetermined conservative limit.

The `-ptst` compiler option increases this limit and can cause a noticeable increase in compile time.

The `-ptstall` option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

The `-noptst` option disables test promotion.

The `PROMOTE_TEST`, `PROMOTE_TEST_ALL` and `NO_PROMOTE_TEST` Fortran compiler directives and `promote_test`, `promote_test_all` and `no_promote_test` C pragmas provide similar functionality on a loop-by-loop basis. See Appendix A, "Compiler directives and pragmas," for more information about these directives.

Using `-ptst`, `-ptstall`, `-promote_test`, `-promote_test_all`, `-PROMOTE_TEST`, or `-PROMOTE_TEST_ALL` will *only* increase the loop replication limit for the test promotion optimization; they will not increase the limit on the total number of replicated loops. This leaves a smaller percentage of the total limit for other loop-replicating options. To increase the total limit, specify the `-mrl` compiler option.

At optimization levels `-O2` and above, the Convex compilers automatically perform these optimizations on Fortran `DO` and hand-rolled loops, and on `for`, `while`, `do-while` and hand-rolled loops in C. Simple and nested loops, and loops with exits, are handled.

---

## Scalar replacement

Scalar replacement involves storing loop-invariant array elements in scalars, which can then be stored in registers by global register allocation. This substantially increases the performance of the loop by eliminating the necessity of accessing the array element in main memory or the cache.

Consider the following Fortran loop:

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

Here,  $A(I)$  is invariant with respect to the  $J$  loop. The compiler can therefore replace  $A(I)$  with a scalar before entering the  $J$  loop, and store the scalar back to  $A(I)$  between iterations of  $I$ . Global register allocation can then assign this scalar to a register for the duration of the iteration, further improving efficiency.

Employing scalar replacement with global register allocation, the compiler generates code equivalent to the following Fortran:

```
DO I = 1, N
  REG = A(I)      ! PLACE A(I) IN REGISTER
  DO J = 1, M
    REG = REG + B(J)
  ENDDO
  A(I) = REG      ! STORE REGISTER BACK TO A(I)
ENDDO
```

Here, the variable "REG" represents a register.

An analogous C example follows:

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

After scalar replacement:

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

Scalar replacement increases performance for this kind of loop by about 50%.

The compiler will not attempt scalar replacement if the loop-invariant array element is aliased or if the array element is contained in conditional code.

At optimization level `-O2` and above, scalar replacement is enabled by default and through use of the `-sr` compiler option; it can be disabled by the `-nsr` compiler option.

---

## Inhibitors of localization

Any of the following conditions can inhibit or prevent data localization:

- Loop-carried dependencies
- Aliased scalar or array variables

- Computed or assigned GOTO statements in Fortran
- Multiple loop entries or exits
- RETURN or STOP statements in Fortran
- Procedure calls
- I/O statements

The following sections discuss these conditions and their effects on data localization.

### Loop-carried dependencies

A loop-carried dependency (LCD) exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration. In some cases, LCDs can inhibit loop interchange, thereby inhibiting localization. Typically, these cases involve array indexes which are offset in opposite directions. The Fortran loop below contains an interchange-inhibiting LCD.

```
DO I = 2, M
  DO J = 1, N
    A(I,J) = A(I-1,J+1) + B(I,J)
  ENDDO
ENDDO
```

C loops can contain similar constructs, but to simplify illustration, we will only consider this Fortran example.

As written, this loop uses  $A(I-1, J+1)$  to compute  $A(I, J)$ . Table 2 shows the sequence in which values of  $A$  are computed for this loop.

**Table 2** Computation sequence of  $A(I, J)$ —original loop

I	J	$A(I, J)$	$A(I-1, J+1)$
2	1	$A(2, 1)$	$A(1, 2)$
2	2	$A(2, 2)$	$A(1, 3)$
2	3	$A(2, 3)$	$A(1, 4)$
...	...	...	...
3	1	$A(3, 1)$	$A(2, 2)$
3	2	$A(3, 2)$	$A(2, 3)$
3	3	$A(3, 3)$	$A(2, 4)$
...	...	...	...

As enumerated in Table 2, the original loop computes the elements of the current row of  $A$  using the elements of the previous row of  $A$ . For all rows except the first (which is never

written), the values contained in the previous row must be written before the current row is computed. This dependency must be honored for the loop to yield its intended results. If a row element of *A* is computed before the previous row element is computed, the result will be incorrect.

Interchanging the *I* and *J* loops yields the following code:

```
DO J = 1, N
  DO I = 2, M
    A(I,J) = A(I-1,J+1) + B(I,J)
  ENDDO
ENDDO
```

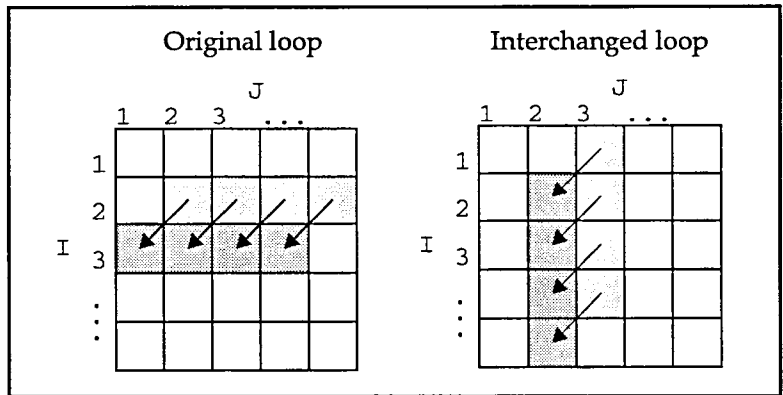
After interchange, the loop computes values of *A* in the sequence shown in Table 3.

**Table 3** Computation sequence of *A*(*I*, *J*)—interchanged loop

<i>I</i>	<i>J</i>	<i>A</i> ( <i>I</i> , <i>J</i> )	<i>A</i> ( <i>I</i> -1, <i>J</i> +1)
2	1	<i>A</i> (2, 1)	<i>A</i> (1, 2)
3	1	<i>A</i> (3, 1)	<i>A</i> (2, 2)
4	1	<i>A</i> (4, 1)	<i>A</i> (3, 2)
...	...	...	...
2	2	<i>A</i> (2, 2)	<i>A</i> (1, 3)
3	2	<i>A</i> (3,2)	<i>A</i> (2,3)
4	2	<i>A</i> (4,2)	<i>A</i> (3,3)
...	...	...	...

Here, the elements of the current column of *A* are computed using the elements of the *next* column of *A*.

The problem here is that columns of *A* are being computed using elements that have not been written yet, which violates the dependency illustrated in Table 2. The element-to-element dependencies in both the original and interchanged loop are illustrated in Figure 14.



**Figure 14** LCDs in original and interchanged loops

The arrows in Figure 14 represent dependencies from one element to another; the arrows point at elements which depend on the elements at the arrows' base. Shaded elements indicate a typical row or column computed in the inner loop; lightly shaded elements have not been computed yet, and darkly shaded elements have already been computed. This figure helps to illustrate the sequence in which the array elements are cycled through by the respective loops: the original loop cycles across all the columns in a row, then moves on to the next row; the interchanged loop cycles down all the rows in a column first, then moves on to the next column.

Interchange is only inhibited by loops that contain dependencies which change when the loop is interchanged. Most LCDs do not fall into this category and thus do not inhibit data localization.

Occasionally the compiler will encounter an apparent LCD. In this case, if it cannot determine whether the LCD actually inhibits interchange, it will conservatively avoid interchanging the loop.

The following Fortran example illustrates this situation:

```
DO I = 1, N
  DO J = 2, M
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

An analogous C example follows.

```
for (j=0; j<n; j++)
  for (i=1; i<m; i++)
    a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

In these examples, if IADD and JADD are either both positive or both negative, the loop contains no interchange-inhibiting

dependency. However, if one and only one of the variables is negative, interchange is inhibited. The compiler has no way of knowing the runtime values of IADD and JADD, so it will avoid interchanging the loop. If you are sure the IADD and JADD will either both be negative or both be positive, you can indicate to the compiler that the loop is free of dependencies using the NO\_LOOP\_DEPENDENCE compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR NO_LOOP_DEPENDENCE (namelist)
```

The no\_loop\_dependence C pragma has the form:

```
#pragma _CNX no_loop_dependence (namelist)
```

Where *namelist* is a comma-delimited list of variables and/or arrays that have no dependencies for the immediately following loop.

The previous Fortran loop can be interchanged when the NO\_LOOP\_DEPENDENCE directive is specified for A and B on the J loop as shown in the following code:

```
DO I = 1, N
C$DIR NO_LOOP_DEPENDENCE (A)
DO J = 2, M
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
ENDDO
ENDDO
```

The no\_loop\_dependence pragma can similarly be used on the C loop:

```
for(i=0;i<n;i++)
#pragma _CNX no_loop_dependence(a)
    for(j=1;j<m;j++)
        a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

If either IADD or JADD (but not both) acquire negative values at runtime, these loops may result in incorrect answers.

### Dependencies and loop fusion

In some cases, loop fusion is also inhibited by simpler dependencies than those that inhibit interchange. Consider the following Fortran example:

```
DO I = 1, N-1
    A(I) = B(I+1) + C(I)
ENDDO
DO J = 1, N-1
    D(J) = A(J+1) + E(J)
ENDDO
```

It would appear that this loop would profit from fusion. Fusing it would yield the following incorrect code:

```
DO ITEMP = 1, N-1
  A (ITEMP) = B (ITEMP+1) + C (ITEMP)
  D (ITEMP) = A (ITEMP+1) + E (ITEMP)
ENDDO
```

This loop produces different answers than the original loops, because the reference to A (ITEMP+1) in the fused loop accesses a value that hasn't been assigned yet, while the analogous reference to A (J+1) in the original J loop accesses a value that was assigned in the original I loop.

An analogous C example follows:

```
for(i=0; i<n-1; i++)
  a[i] = b[i+1] + c[i];
for(j=0; j<n-1; j++)
  d[j] = a[j+1] + e[j];
```

After fusion:

```
for(itemp=0; itemp<n-1; itemp++) {
  a[itemp] = b[itemp+1] + c[itemp];
  d[itemp] = a[itemp+1] + e[itemp];
}
```

### Aliasing

An *alias* is an alternate name for some object. Aliasing occurs in a program when two or more names are attached to the same memory location. Aliasing is typically caused in Fortran by use of the EQUIVALENCE statement and in C by use of pointers. Passing identical actual arguments into different dummy arguments in a Fortran subprogram can also cause aliasing, as can passing the same address into different pointers in a C function.

Aliasing interferes with data localization because it can mask LCDs, as shown in the following Fortran example, where the arrays A and B have been equivalenced.

```
INTEGER A(100,100), B(100,100), C(100,100)
EQUIVALENCE (A, B)
.
.
.
DO I = 1, N
  DO J = 2, M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
```

This loop has the same problem as the loop used to demonstrate LCDs in the previous section; since A and B refer to the same array, the loop contains an LCD on A, which prevents interchange and thus interferes with localization.

The C equivalent of this loop follows. Keep in mind that C stores arrays in row-major order, which requires different subscripting to access the same elements.

```
int a[100][100], c[100][100], i, j;
int (*b)[100];
b = a;
.
.
.
for(i=1; i<n; i++){
    for(j=0; j<m; j++){
        a[j][i] = b[j+1][i-1] + c[j][i];
    }
}
```

C has nothing analogous to Fortran's EQUIVALENCE statement, but arrays can be effectively equivalenced through the use of pointers, as shown.

Passing the same address into different dummy procedure arguments can yield the same result. Fortran passes arguments by reference while C passes them by value, but pass-by-reference can be simulated in C by passing the argument's address into a pointer in the receiving procedure.

The following Fortran program exhibits the same aliasing problem as the previous example, but the alias is created by passing the same actual argument into different dummy arguments.

## Caution

The following code violates the Fortran standard.

```
.  
. .  
. .  
CALL ALI (A,A,C)  
. .  
SUBROUTINE ALI (A,B,C)  
INTEGER A(100,100), B(100,100), C(100,100)  
DO J = 1, N  
  DO I = 2, M  
    A(I,J) = B(I-1,J+1) + C(I,J)  
  ENDDO  
ENDDO  
. .  
. .  
. .
```

The following code shows the same argument-passing problem in C (this code is legal ANSI C).

```
.  
. .  
. .  
ali (&a, &a, &c);  
. .  
. .  
void ali (a,b,c)  
int a[100][100], b[100][100], c[100][100];  
{  
  int i,j;  
  for (j=0; j<n; j++){  
    for (i=1; i<m; i++){  
      a[j][i] = b[j+1][i-1] + c[j][i];  
    }  
  }  
}
```

### Multiple loop entries or exits

Loops that contain multiple entries or exits inhibit data localization because they cannot safely be interchanged. Extra loop entries are usually created when a loop contains a branch destination. Extra exits are more common; they are often created in C using the `break` statement and in Fortran using the `GOTO` statement.

Consider the following C code:

```
for(j=0;j<n;j++){
  for(i=0;i<m;i++){
    a[i][j] = b[i][j] + c[i][j];
    if(a[i][j] == 0) break;
    .
    .
    .
  }
}
```

Interchanging this loop would change the order in which the values of `a` are computed; the original loop computes a column-by-column, whereas the interchanged loop would compute it row-by-row. This means that the interchanged loop may hit the `break` statement and exit after computing a different set of elements than the original loop. Interchange therefore may cause the results of the loop to differ and must be avoided.

A similar loop construct written in Fortran follows:

```
DO J = 1, M
  DO I = 1, N
    A(I,J) = B(I,J) + C(I,J)
    IF(A(I,J) .EQ. 0) GOTO 50
    .
    .
    .
  ENDDO
ENDDO
.
.
.
50 CONTINUE
```

Again, the order of computation changes if the loops are interchanged.

### **RETURN or STOP statements in Fortran**

Like loops with multiple exits, `RETURN` and `STOP` statements in Fortran inhibit localization because they inhibit interchange. If a loop containing a `RETURN` or `STOP` is interchanged, its order of computation may change, giving wrong answers.

### **Computed or assigned GOTO statements in Fortran**

When the Fortran compiler encounters a computed or assigned `GOTO` statement in an otherwise interchangeable loop, it cannot always determine whether the branch destination is within the

loop. Since an out-of-loop destination would be a loop exit, these statements often prevent loop interchange and therefore data localization.

### Procedure calls

The standard Convex compilers are unaware of the side effects of most procedures, and therefore cannot determine whether they might interfere with loop interchange. These side effects may include data dependencies involving loop arrays, aliasing (as described in the "Aliasing" section), and processor data cache usage that conflicts with the loop's usage of the cache, rendering useless any data localization optimizations performed on the loop.

### I/O statements

The order in which values are read into or written from a loop may change if the loop is interchanged, so I/O statements inhibit interchange and therefore data localization. For example, consider the following Fortran code:

```
DO I = 1, 4
  DO J = 1, 4
    READ *, IA(I,J)
  ENDDO
ENDDO
```

Given a data stream consisting of alternating zeros and ones (0,1,0,1,0,1...), the contents for  $A(I, J)$  for both the original loop and the interchanged loop are shown in Figure 15.

		Original loop				Interchanged loop			
		J				J			
		1	2	3	4	1	2	3	4
I	1	0	1	0	1	1	1	1	1
	2	0	1	0	1	0	0	0	0
	3	0	1	0	1	1	1	1	1
	4	0	1	0	1	0	0	0	0

Figure 15 Values read into array A

C loops exhibit the same limitations. A C example that produces the data patterns shown in Figure 15 follows:

```
for (i=1; i<5; i++)
    for (j=1; j<5; j++)
        scanf ("%d", &ia[i][j]);
```

---

## Preventing data localization

The `SCALAR` directive allows you to prevent all loop-reordering transformations on the immediately following loop. In Fortran, it has the following form:

```
C$DIR SCALAR
```

In C it has the following form:

```
#pragma _CNX scalar
```

---

## -O3 Level optimizations

The primary goal of `-O3` optimizations is *parallelization*.

Parallelization divides a program into threads. A *thread* is a single flow of control within a process. It can be a unique flow of control that performs a specific function, or one of several instances of a flow of control, each of which is operating on a unique data set.

The Convex SPP Series compilers find parallelism at the loop level and generate parallel code that will automatically run on as many processors as are available at runtime. Normally, these are all the processors of the subcomplex on which your program is running. You can specify a smaller number of processors via the `mpa` utility. Refer to the `mpa (1)` man page for more information. You can also specify a smaller number of processors using the `loop_parallel(max_threads=m)` or `prefer_parallel(max_threads=m)` compiler directives and pragmas, as described in chapters 4 and 6.

Automatic parallelization is only useful when your program is written using the shared memory paradigm or a combination of message passing and shared memory. If you are writing your program entirely under the message passing paradigm, you must explicitly handle parallelism yourself as discussed in Chapter 7, "Message passing programming." Such pure message passing programs should be compiled at optimization level `-O2` or lower.

---

## Basic operation

Parallelism can exist at the loop level and the task level. Convex SPP Series compilers automatically exploit loop-level parallelism.

You can easily identify task-level parallelism using the `begin_tasks`, `next_task` and `end_tasks` directives and pragmas, as discussed in the “Task parallelization” section of Chapter 4, “Basic shared memory programming.” These directives and pragmas allow the compiler to run identified sections of code in parallel.

Loop-level parallelism involves dividing a loop up into several smaller iteration spaces and parceling these out to be run simultaneously on the available processors.

## Note

**Only loops with iteration counts determinable prior to loop invocation at runtime are candidates for parallelization. Loops with iteration counts that depend on values or conditions calculated within the loop cannot be parallelized by any means.**

Consider the following Fortran code:

```
PROGRAM PARAXPL
.
.
.
DO I = 1, 1024
  A(I) = B(I) + C(I)
.
.
.
ENDDO
```

Assuming the `I` loop doesn't contain any parallelization-inhibiting code, this program can be parallelized to run on 8 processors by running 128 iterations per processor (1024 iterations divided by 8 processors = 128 iterations each). One processor would run the loop for `I = 1` to 128; the next would run `I = 129` to 256, and so on. The loop could similarly be parallelized to run on any number of processors, with each one taking its appropriate share of iterations, but at a certain point adding more processors will not improve performance. The compiler generates code that will run on as many processors as are available, but the dynamic selection optimization (described in the “Dynamic selection” section) ensures that parallel code is generated only if it is profitable to do so. If the number of available processors does not evenly divide the number of iterations, some processors will perform fewer iterations than others.

On Exemplar, shared memory programs run as a collection of threads on multiple processors. When a program starts, a separate execution thread is created on each of the processors of the subcomplex on which the program is running. Each of these threads is identified by SPP-UX by a unique *kernal* thread ID. All

but one of these threads is then idle; the nonidle thread is known as thread 0, and this thread runs all of the serial code in the program. *Spawn* thread IDs are assigned only to nonidle threads when they are spawned—that is, when thread 0 encounters parallelism and “wakes up” other idle threads to execute the parallel code. Spawn thread IDs are consecutive. They range from 0 to  $nt-1$ , where  $nt$  is the number of threads spawned as a result of the spawn operation; this operation defines the current *spawn context*. The spawn context is the loop or task list that initiates the spawning of the threads. Spawn thread IDs are valid only within a given spawn context.

Therefore, the idle threads are not assigned spawn thread IDs at the time of their creation. When thread 0 encounters a parallel loop or task, it spawns the other threads, signaling them to begin execution. The threads then become active, acquire spawn thread IDs, run until their portion of the parallel code is finished, and go idle once again, as shown in Figure 16. Machine loading does not affect the number of threads spawned, but it may affect the order in which the threads in a given spawn context complete. SPP-UX ensures that, regardless of the machine load, all child threads of a given process execute to the completion of their spawn context before that process’s thread 0 continues.

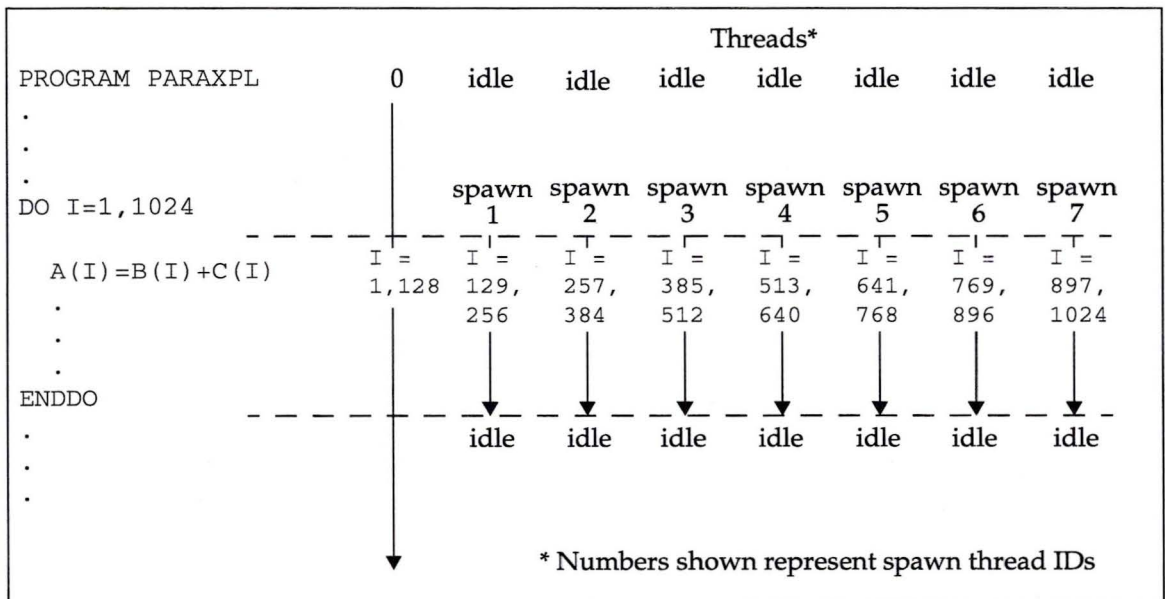


Figure 16 Thread activity: one-dimensional parallelism

## Note

This example is greatly simplified for illustrative purposes. Various loop transformations can affect the manner in which a loop is parallelized.

To actually implement this, the compiler transforms the loop in a manner similar to strip mining. However, unlike the strip mining described in the “-O2 Level optimizations” section, the outer loop is conceptual; since the strips will be executing on different CPUs, there is no processor to run an outer loop like the one created in traditional strip mining.

Instead, the loop is transformed such that the starting and stopping iteration values are variables that are determined at runtime based on how many threads are available and which thread is running the strip in question.

Consider our previous Fortran example written for an unspecified number of iterations.

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

The code shown in Figure 17 is a conceptual representation of the transformation the compiler performs on this example when it is compiled for parallelization, assuming that  $N \geq \text{NumThreads}$ . For  $N < \text{NumThreads}$ , the compiler uses  $N$  threads, assuming there is enough work in the loop to justify the overhead of parallelizing it. If  $\text{NumThreads}$  is not an integral divisor of  $N$ , some threads will do fewer iterations than others.

```
For each available thread do:
  DO I=ThrdID*(N/NumThreads)+1,ThrdID*(N/NumThreads)+N/NumThreads
    A(I) = B(I) + C(I)
  ENDDO
ENDDO
```

**Figure 17** Conceptual strip mine for parallelization

$\text{NumThreads}$  is the number of available threads.  $\text{ThrdID}$  is the ID number of the thread this particular loop will run on, which is between 0 and  $\text{NumThreads}-1$ . A unique  $\text{ThrdID}$  is assigned to each thread, and the  $\text{ThrdIDs}$  are consecutive. So, for  $\text{NumThreads} = 8$ , as in , 8 loops would be spawned, with  $\text{ThrdIDs} = 0$  through 7. These 8 loops are illustrated in Figure 18.

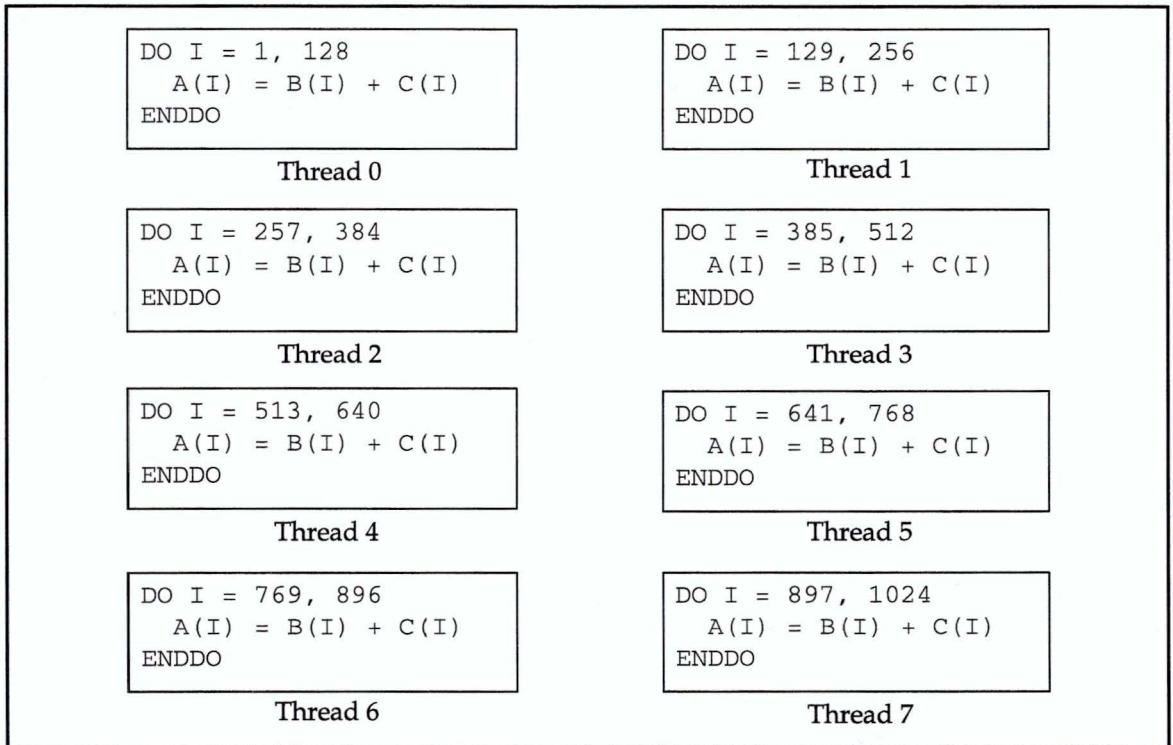


Figure 18 Parallelized loop

## Note

The strip-based parallelism described here is the default. Stride-based parallelism is possible through use of the `PREFER_PARALLEL` and `LOOP_PARALLEL` compiler directives and pragmas, which are described in Chapter 4, “Basic shared memory programming.”

In these examples, the data being manipulated within the loop is disjoint; that is, no two threads attempt to write the same data item. If two parallel threads attempt to update the same storage location, their actions must be synchronized. This is discussed further in Chapter 4 “Basic shared memory programming.”

### Idle thread states

Idle threads can be suspended or spin-waiting. Suspended threads release control of the processor; spin-waiting threads repeatedly check an encached global semaphore which indicates whether or not they have code to execute. Obviously, this prevents any other process from gaining control of the CPU and can severely degrade multiprocess performance. On the other hand, waking a suspended thread takes substantially longer than activating a spin-waiting thread. Therefore, by default, idle

threads spin-wait briefly after creation or a join, then suspend themselves if no work is received. This spin-wait time can be changed by using the `cps_wait_attr()` function, which is described in Appendix D.

When threads are suspended, SPP-UX may schedule threads of another process on their CPUs in order to balance the machine load. However, threads have an *affinity* for their original CPUs; SPP-UX attempts to schedule the threads that have been swapped out or suspended to their original CPUs in order to exploit the presence of any data encached during the thread's last timeslice. This affinity is only realized if the original CPU is available; otherwise, the thread is assigned to the first CPU to become available within the hypernode on which it was spawned.

---

## Fortran 90 constructs

Fortran 90 array expressions are translated into loops by the compiler, and parallelism in these loops will be automatically exploited. For example, the following Fortran 90 statement:

```
X(1:M:2, 1:N) = Y(2:M+1:2, 2:N+1)
```

is translated by the compiler into a loop similar to the following:

```
DO J = 1, M, 2
  DO I = 1, N
    X(J,I) = Y(J+1,I+1)
  ENDDO
ENDDO
```

which can then be automatically parallelized.

Masked array assignments are similarly parallelized. Consider the following `WHERE` statement:

```
REAL DATA(1000), LIMIT
LOGICAL NORMAL(1000)
.
.
.
WHERE (DATA .LE. LIMIT) NORMAL = .TRUE.
```

The compiler translates the `WHERE` statement into a loop similar to the following:

```
DO I = 1, 1000
  IF (DATA(I) .LE. LIMIT) NORMAL(I) = .TRUE.
ENDDO
```

which can then be automatically parallelized.

---

## Parallel optimizations

Simple loops can be parallelized without the need for extensive  $-O2$  transformations, as shown in the “Basic operation” section. However, most loop transformations, if they are applicable to the loop in question, can aid parallelization in some way. For instance, loop interchange orders loops such that the innermost loop best exploits the processor data cache, and the outermost loop is the most efficient loop to parallelize. Loop blocking similarly aids parallelization by maximizing cache data reuse on each of the processors that the loop runs on, and by ensuring that each processor is working on nonoverlapping array data.

All optimizations performed at lower optimization levels are performed at optimization level  $-O3$ .

### Dynamic selection

The compiler has no way of determining how many processors will be available to run compiled code; therefore, it must generate both serial and parallel code for loops that can be parallelized. Replicating the loop in this manner is called *cloning*, and the resulting versions of the loop are called *clones*.

It is not always profitable, however, to run the parallel clone when multiple processors are available. A great deal of overhead is involved in executing parallel code. This overhead includes the time it takes to spawn parallel threads, to privatize any variables used in the loop that must be privatized, and to join the parallel threads when they complete their work.

Convex SPP Series compilers use a powerful form of dynamic selection known as *workload-based dynamic selection*. When a loop’s iteration count is available at compile time, workload-based dynamic selection determines the profitability of parallelizing the loop and only writes a parallel version to the executable if it is profitable to do so. Omitting the parallel version from the executable when it will never be needed enhances performance further by eliminating the runtime decision as to which version to use.

The power of dynamic selection becomes more apparent when the loop’s iteration count is unknown at compile time. In this case, the compiler generates code that, at runtime, compares the amount of work performed in the loop nest (given the actual iteration counts) to the parallelization overhead for the available number of processors and runs the parallel version of the loop only if it is profitable to do so.

Workload-based dynamic selection is enabled by default at optimization level `-O3` on SPP Series machines. The `-nds` compiler option can be used to disable dynamic selection; the `-ds` compiler option can be used to override `-nds`. When dynamic selection is disabled, the compiler assumes that it is profitable to parallelize all parallelizable loops, and generates both serial and parallel clones for them. In this case the parallel version is run if there are multiple processors at runtime regardless of the profitability of doing so.

You can enable workload-based dynamic selection for selected loops by using the `dynsel` compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR DYNSEL[(THREAD_TRIP_COUNT = n | NODE_TRIP_COUNT = m)]
```

The C pragma has the following form:

```
#pragma _CNX dynsel[(thread_trip_count = n | node_trip_count = m)]
```

Where `thread_trip_count` and `node_trip_count` are optional attributes used to specify threshold iteration counts. If `thread_trip_count = n` is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. If `node_trip_count = m` is specified, the serial version of the loop is run if the iteration count is less than *m*; otherwise, the node-parallel version is run. *n* and *m* must be compile-time constants.

The `thread_trip_count` attribute cannot be used on loops which also specify the `loop_parallel(nodes)` directive or pragma. Similarly, the `node_trip_count` attribute cannot be used on loops which also specify the `loop_parallel(threads)` directive or pragma. These combinations are contradictory.

These directives can be used to specify dynamic selection for specific loops in programs compiled using the `-nds` option, or to provide trip count information for specific loops in programs compiled with dynamic selection enabled.

You can disable dynamic selection for selected loops by using the `no_dynsel` compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR NO_DYNSEL
```

The C pragma has the following form:

```
#pragma _CNX no_dynsel
```

This directive or pragma can be used to disable dynamic selection on specific loops in programs compiled with dynamic selection enabled.

As with all optimizations that replicate loops, the number of new loops created when the compiler performs dynamic selection is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler will issue an advisory and workload-based dynamic selection will be disabled for any following loops. To increase the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mr1` compiler option.

---

## Inhibitors of parallelization

Most constructs that inhibit data localization also inhibit parallelization. Specifically, these are

- Loop-carried dependencies
- Aliased scalar or array variables
- Multiple loop entries or exits
- Procedure calls
- I/O statements

Most of these items inhibit parallelization for the same reasons they inhibit localization. An exception to this is that more categories of loop carried dependencies can inhibit parallelization than data localization as described in the following sections.

### Loop-carried dependencies

The specific loop-carried dependencies that inhibit data localization represent a very small portion of all loop-carried dependencies. A much broader set of LCDs, including those that inhibit data localization, can inhibit parallelization.

LCDs fall into three categories: *forward* LCDs, *backward* LCDs and *output* LCDs. The LCD that inhibits localization is a combination of these. All of these LCDs inhibit parallelization.

#### Forward LCDs

A forward LCD exists when one iteration references a variable whose value is assigned on a later iteration. The Fortran loop below contains a forward LCD on the array *A*.

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

In this example, the first iteration assigns a value to *A*(1) and references *A*(2). The second iteration assigns a value to *A*(2) and references *A*(3). The reference to *A*(*I*) depends on the fact that the *I*+1th iteration, which assigns a new value to *A*(*I*), has not

yet executed. Forward LCDs inhibit parallelization because if the loop is broken up to run on several processors, when  $I$  reaches its terminal value on one processor,  $A(I+1)$  will usually have already been computed by another processor (it is, in fact, the first value computed by another processor). Since the calculation depends on  $A(I+1)$  being uncomputed, this would produce wrong answers.

An analogous C loop follows:

```
for(i=0; i<n-1; i++)
    a[i] = a[i+1] + b[i];
```

### Backward LCDs

A backward LCD exists when one iteration references a variable whose value was assigned on an earlier iteration. The Fortran loop below contains a backward LCD on the array  $A$ .

```
DO I = 2, N
    A(I) = A(I-1) + B(I)
ENDDO
```

Here, each iteration assigns a value to  $A$  based on the value assigned to  $A$  in the previous iteration. If  $A(I-1)$  has not been computed before  $A(I)$  is assigned, wrong answers will result. Backward LCDs inhibit parallelism because if the loop is broken up to run on several processors,  $A(I-1)$  will not have been computed for the first iteration of the loop on every processor except the processor running the chunk of the loop containing  $I = 1$ .

An analogous C loop follows:

```
for(i=1; i<n; i++)
    a[i] = a[i-1] + b[i];
```

### Output LCDs

An output LCD exists when the compiler cannot determine whether an array subscript contains the same values between loop iterations. The Fortran loop below contains a potential output LCD on the array  $A$ .

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

Here, if any referenced elements of  $J$  contain the same value, the same element of  $A$  might be assigned several different elements of  $B$ . In this case, as this loop is written, any  $A$  elements that are assigned more than once should contain the final assignment at

the end of the loop. If the loop is run in parallel, however, this cannot be guaranteed.

An analogous C loop follows:

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

### Apparent LCDs

As at optimization level `-O3`, the compiler will not parallelize loops containing apparent LCDs rather than risk wrong answers by doing so.

If you are sure that a loop with an apparent LCD is safe to parallelize, you can indicate this to the compiler using the `NO_LOOP_DEPENDENCE` directive or pragma, which is explained in the “`-O2 Level optimizations`” section.

The following Fortran example illustrates a `NO_LOOP_DEPENDENCE` directive being used on the output LCD example presented previously.

```
C$DIR NO_LOOP_DEPENDENCE (A)
      DO I = 1, N
          A(J(I)) = B(I)
      ENDDO
```

This effectively tells the compiler that no two elements of `J` are identical, so there is no output LCD and the loop is safe to parallelize. If any of the `J` values are identical, wrong answers could result.

Use of the `no_loop_dependence` pragma is illustrated in the following C example.

```
#pragma _CNX no_loop_dependence (a)
for (i=0; i<n; i++)
    a[j[i]] = b[i];
```

---

## Reductions

In many cases, the compiler can recognize and parallelize loops containing a special class of dependency known as a reduction. In general, a reduction has the form:

$$X = X \text{ operator } Y$$

where `X` is a variable not assigned or used elsewhere in the loop, `Y` is a loop constant expression not involving `X`, and *operator* is `+`, `-`, `*`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`

The compiler also recognizes reductions of the form:

$$X = \text{function}(X, Y)$$

where  $X$  is a variable not assigned or referenced elsewhere in the loop,  $Y$  is a loop constant expression not involving  $X$ , and *function* is the intrinsic MAX function or intrinsic MIN function.

Reductions commonly appear in the form of sum operations, as shown in the following Fortran example.

```
DO I = 1, N
  A(I) = B(I) + C(I)
  .
  .
  .
  ASUM = ASUM + A(I)
ENDDO
```

Assuming this loop does not contain any parallelization-inhibiting code, the compiler would automatically parallelize it. The code generated to accomplish this creates temporary, thread-private copies of ASUM for each thread that will be running the loop. When each parallel thread completes its portion of the loop, it updates the global ASUM variable in a section of code that prevents other threads from accessing ASUM simultaneously.

The optimization report would indicate that ASUM was privatized and its value saved for use after the loop.

An analogous C example follows.

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

---

## Preventing parallelization

Parallelization can be disabled on a loop basis by using the NO\_PARALLEL directive or pragma. The Fortran directive has the following form:

```
C$DIR NO_PARALLEL
```

The C pragma has the following form:

```
#pragma _CNX no_parallel
```

You can use these directives to prevent parallelization of the loop that immediately follows them. Only parallelization is inhibited; all other loop optimizations will still be applied. The following Fortran example illustrates the use of this directive.

```
      DO I = 1, 1000
C$DIR   NO_PARALLEL
      DO J = 1, 1000
          A(I,J) = B(I,J)
      ENDDO
ENDDO
```

In this example, parallelization of the J loop is prevented. The I loop can still be parallelized.

An analogous C example follows:

```
for(i=0;i<1000;i++)
#pragma _CNX no_parallel
    for(j=0;j<1000;j++)
        a[i][j] = b[i][j];
```

The `-noautopar` compiler option is available to disable automatic parallelization but will allow parallelization of directive- or pragma-specified loops. Refer to Chapter 4, “Basic shared memory programming,” for more information on `-noautopar`.

---

## Other parallelization directives and pragmas

Several directives and pragmas are available to allow you to manually control certain aspects of loop parallelization, and to parallelize tasks outside of loops. These directives are:

`prefer_parallel`

Requests parallelization of the immediately following loop; accepts attributes for node- and thread-parallelism, strip-length adjustment, maximum number of threads, and ordered execution. The compiler handles data privatization and will not parallelize the loop if it is not safe to do so.

`loop_parallel`

Forces parallelization of the immediately following loop. Accepts the same attributes as `PREFER_PARALLEL`, but requires you to manually privatize loop data and synchronize data dependencies.

`begin_tasks`, `next_task` and `end_tasks`

Allow you to parallelize consecutive blocks of code outside of loops. Accepts attributes for node- and thread-parallelism, ordered execution, and maximum number of threads.

`critical_section`, `end_critical_section`

Allow you to isolate nonordered manipulations of a shared variable within the loop. Only one parallel thread can execute the code contained in the critical section at a time, eliminating possible contention.

`ordered_section`, `end_ordered_section`

Allow you to isolate dependencies within a loop so that code contained within the ordered section executes in iteration order. Only useful when used with the `loop_parallel(ordered)` or `prefer_parallel(ordered)` directives or pragmas.

These directives and pragmas are discussed in detail in Chapter 4, "Basic shared memory programming," and Chapter 6, "Advanced shared memory programming."



---

# Basic shared memory programming

# 4

This chapter discusses programming techniques that allow you to increase code efficiency with minimal effort.

---

## Simple manual loop and task parallelization

The Convex SPP Series compilers will automatically exploit strip-based loop parallelism in loops that are clearly dependency-free, as described in Chapter 3, “Compiler optimizations”. The `prefer_parallel`, `loop_parallel`, and `parallel` directives and pragmas allow you to increase parallelization opportunities and to manually control many aspects of parallelization.

The compiler cannot automatically locate task parallelism, but the tasking directives mentioned in Chapter 3, “Compiler optimizations,” and discussed here allow you to specify consecutive blocks of code that can be run in parallel.

The subsections that follow discuss specifying simple, unordered loop and task parallelism using the `prefer_parallel`, `loop_parallel`, and `begin_tasks/next_task/end_tasks` directives and pragmas. Critical sections that do not rely on ordered execution are also covered here. Any necessary variable privatization is provided by the `loop_private`, `task_private` and `parallel_private` directives and pragmas, which are described in detail in the “Loop- and task-specific data privatization” section of this chapter.

For a detailed discussion of ordered parallelism, parallel synchronization, and the effective use of memory classes, refer to Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared memory programming.”

---

## Loop parallelization

This section discusses simple uses of the `prefer_parallel` and `loop_parallel` directives and pragmas, which, when specified, apply to the immediately following loop. The data privatization necessary when using `loop_parallel` is implemented in this chapter's examples using the `loop_private` directive discussed on page 129. Manual data privatization using memory classes is discussed in Chapter 5, "Memory classes," and Chapter 6, "Advanced shared memory programming."

## Note

**These directives should only be used on Fortran DO and C for loops that have iteration counts determinable prior to loop invocation at runtime.**

`loop_parallel` and `prefer_parallel` generally take the same attributes. In Fortran, these directives have the following form:

```
C$DIR PREFER_PARALLEL[ (attribute-list) ]
```

and

```
C$DIR LOOP_PARALLEL[ (attribute-list) ]
```

In C, they have the following form:

```
#pragma _CNX prefer_parallel[ (attribute-list) ]
```

and

```
#pragma _CNX loop_parallel(ivar = indvar[, attribute-list])
```

where the optional *attribute-list* can contain one of the following case-insensitive attributes:

`threads`

Causes thread-parallelism. This is the default.

`nodes`

Causes node-parallelism.

`chunk_size = n`

Divides the loop into chunks of *n* or fewer iterations, and distributes the chunks round-robin to the processors. *n* must be a constant integer which has a value at compile time.

`max_threads = m`

Allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.

ordered

Causes ordered invocation of each loop iteration; provides no automatic synchronization. Designed for use with the `loop_parallel` directive and `pragma`, on loops containing ordered sections.

ordered, nodes

Causes ordered invocation of each iteration across hypernodes.

ordered, threads

Causes ordered invocation of each iteration across threads.

nodes, `chunk_size = n`

Node-parallelism by chunks.

threads, `chunk_size = n`

Thread-parallelism by chunks.

`chunk_size = n, max_threads = m`

Chunk parallelism on no more than  $m$  threads.

ordered, `max_threads = m`

Ordered parallelism on no more than  $m$  threads.

nodes, `max_threads = m`

Node-parallelize on no more than  $m$  nodes; this starts one thread per node on no more than  $m$  hypernodes.

threads, `max_threads = m`

Thread-parallelism on no more than  $m$  threads.

ordered, nodes, `max_threads = m`

Ordered node-parallelism on no more than  $m$  hypernodes.

ordered, threads, `max_threads = m`

Ordered thread-parallelism on no more than  $m$  threads.

nodes, `chunk_size = n, max_threads = m`

Node-parallelize by chunks of size  $n$  on no more than  $m$  hypernodes.

threads, `chunk_size = n, max_threads = m`

Thread-parallelize by chunks of size  $n$  on no more than  $m$  threads.

`ivar = indvar`

Use with `loop_parallel` only. Specifies the primary loop induction variable; optional in Fortran, but required in C.

## Note

The values of  $n$  and  $m$  must be compile time constants for all of the above attributes in which they appear.

Only the listed combinations of attributes are allowed; however, in such combinations the attributes can be listed in any order. The `loop_parallel` C pragma requires the `ivar = indvar` attribute, which specifies the primary loop induction variable. If this is not present, the compiler will issue a warning and the pragma will be ignored. `ivar` should specify only the primary induction variable; any other loop induction variables should be a function of this variable and should be declared `loop_private`. In Fortran, `ivar` is optional for DO loops; if not provided, the compiler will pick the primary induction variable for the loop. `ivar` is required for DO WHILE and hand rolled loops in Fortran. `prefer_parallel` does not require `ivar`, and the compiler will issue an error if it encounters this combination.

The optional `threads` attribute causes parallelization across threads; this is the default for `loop_parallel`. If the `threads` attribute appears in a parallelization directive on the outermost loop in a nest, the loop will go parallel on all the threads available to the process. If the `threads` attribute appears in a parallelization directive nested within a node-parallel construct, the specified loop will go thread-parallel on the processors of each parallel hypernode.

The optional `nodes` attribute causes parallelization across hypernodes. In this case, a single thread on each available hypernode will execute a portion of the specified loop. A `loop_parallel(nodes)` directive or pragma disables parallelization analysis for any loops inside the node-parallel loop; in this case if you want to exploit nested parallelism, you must explicitly indicate the thread parallel construct contained within the node-parallel loop. A node-parallel construct cannot exist inside a thread-parallel construct.

The optional `chunk_size` attribute specifies a number of iterations by which to strip mine the loop for parallelization. If this attribute is present alone or with the `threads` attribute,  $n$  or fewer loop iterations are distributed round-robin to each available thread. If `chunk_size` is combined with the `nodes` attribute, the chunks are distributed to one thread per available hypernode. If the number of threads does not evenly divide the number of iterations, some threads will perform one less chunk than others.  $n$  must be an integer constant or constant expression determinable at compile time.

This stride-based parallelism differs from the default strip-based parallelism described in Chapter 3, "Compiler optimizations," which divides the loop's iterations into a number of contiguous

chunks equal to the number of available threads, and each thread computes one chunk. The `chunk_size` attribute allows each thread to do several noncontiguous chunks. Specifying `chunk_size = (number of iterations/number of threads)` is equivalent to default strip mining for parallelization.

Consider the following Fortran example, which uses the `PREFER_PARALLEL` directive, but applies to `LOOP_PARALLEL` as well.

```
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 4)
DO I = 1, 100
    A(I) = B(I) + C(I)
ENDDO
```

In this example, the loop is parallelized by parcelling out chunks of 4 iterations to each available thread. Figure 19 uses Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming 8 available threads.

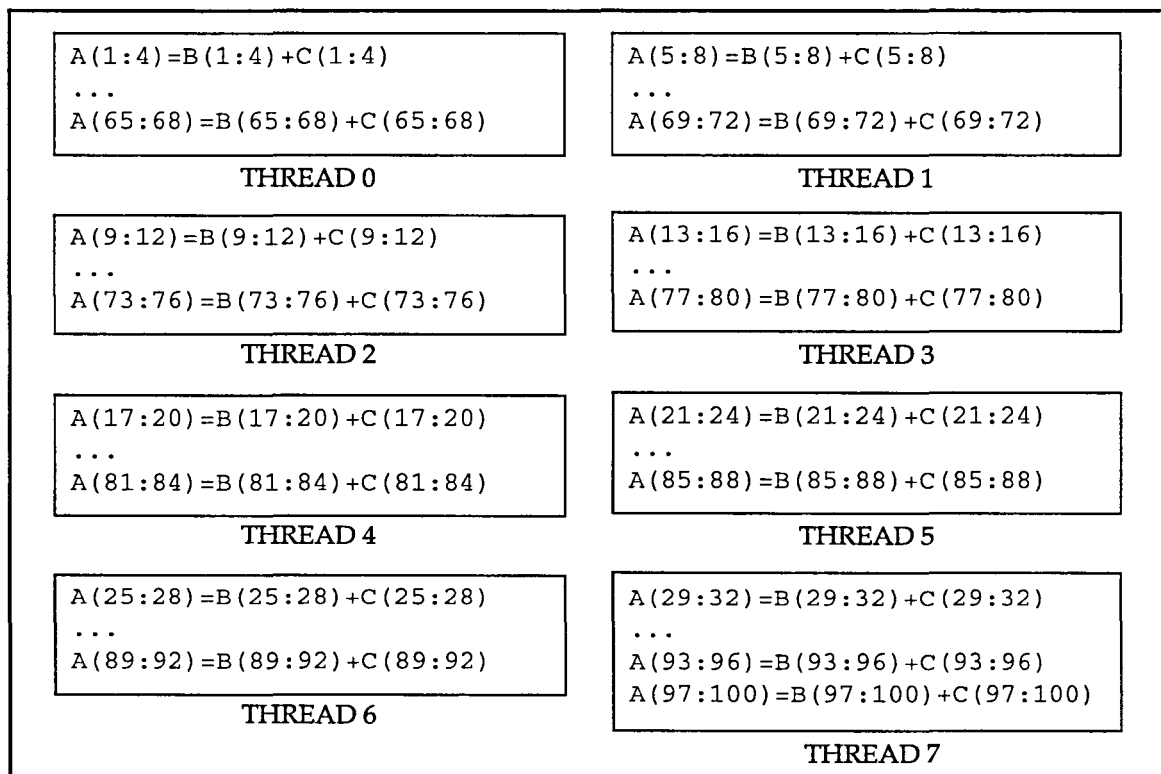


Figure 19 Stride-parallelized loop

Figure 19 shows that the 100 iterations of `I` are parceled out in chunks of 4 iterations to each of the 8 available threads; after the

chunks are distributed evenly to all threads, there is one chunk left over (iterations 97:100), which executes on thread 7.

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(i=0;i<100;i++)
    a[i] = b[i] + c[i];
```

The `chunk_size` attribute is most useful on loops in which the amount of work increases or decreases as a function of the iteration count. The following Fortran example shows such a loop. Again, `PREFER_PARALLEL` is used here, but the concept applies to `LOOP_PARALLEL` also.

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO J = 1,N
        DO I = J, N
          A(I,J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the work of the `I` loop decreases as `J` increases. By specifying a `chunk_size` for `J`, we more evenly balance the load across the threads executing the loop. If this loop was strip mined in the traditional manner, the amount of work contained in the strips would decrease with each successive strip; the threads performing early iterations of `J` would do substantially more work than those performing later iterations.

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(j=0;j<n;j++)
    for(i=j;i<n;i++) {
        a[i][j] = ...
        .
        .
        .
    }
```

The `max_threads = m` attribute restricts execution of the specified loop to no more than `m` threads if specified alone or with the `threads` attribute; if specified with the `nodes` attribute execution is restricted to `m` nodes running one thread each. If specified with the `chunk_size = n` attribute, the chunks are parallelized across no more than `m` threads. `max_threads` is useful when you know the maximum number of threads your loop will run on efficiently.

The `ordered` directive causes the iterations of the loop to be initiated in loop order across the processors. It is only useful in loops with manually-synchronized dependencies, so it is only useful with the `loop_parallel` directive. To achieve ordered parallelism, dependencies must be synchronized within ordered sections, such as those constructed using the `ordered_section` and `end_ordered_section` directives. Using `loop_parallel(ordered)` and its associated synchronization directives is covered in Chapter 6, "Advanced shared memory programming."

### **`prefer_parallel`**

The `prefer_parallel` directive and `pragma` cause the compiler to parallelize the immediately following loop if it is free of dependencies and other parallelization inhibitors. The compiler automatically privatizes any loop variables that must be privatized. On multihypernodal subcomplexes, when `prefer_parallel` is specified without a `nodes` or `threads` attribute, the compiler will determine if opportunities for nested parallelism exist within the loop and parallelize it across nodes or threads appropriately. If the `nodes` attribute is specified, the compiler will automatically find and exploit thread parallelism within the loop. `prefer_parallel` requires less manual intervention and is less forceful than the `loop_parallel` directive and `pragma`.

`prefer_parallel` can also be used to indicate the preferred loop in a nest to parallelize, as shown in the following Fortran example:

```
      DO J = 1, 100
C$DIR   PREFER_PARALLEL
          DO I = 1, 100
              .
              .
              .
          ENDDO
      ENDDO
```

In this example, `PREFER_PARALLEL` causes the compiler to choose the innermost loop for parallelization, provided it is free of dependencies. `PREFER_PARALLEL` does not inhibit loop interchange.

An analogous C example follows:

```
for (j=0; j<100; j++)
    #pragma _CNX prefer_parallel
    for (i=0; i<100; i++) {
        .
        .
        .
    }
```

The ordered attribute should not be used in a `prefer_parallel` directive, as it is only useful if the loop contains synchronized dependencies, and `prefer_parallel` will not parallelize a loop containing any dependencies. This attribute is useful in the `loop_parallel` directive, as described in Chapter 6, "Advanced shared memory programming."

### **loop\_parallel**

The `loop_parallel` directive forces parallelization of the immediately following loop. The compiler does not check for data dependencies, perform variable privatization, or perform parallelization analysis; you must synchronize any dependencies manually, manually privatize loop data as necessary, and if you wish to exploit nested parallelism, you must explicitly do so using directives or pragmas to define thread-parallel constructs within the node-parallel loop. In absence of a `nodes` or `threads` attribute, `loop_parallel` defaults to thread parallelization.

The "Critical sections" section contains an example of using `loop_parallel` to parallelize a loop with a dependency; the dependency is manually handled in a critical section.

The `threads`, `nodes`, `chunk_size` and `max_threads` attributes and combinations of these attributes have exactly the same affect as explained for `prefer_parallel`. `loop_parallel(ordered)` is useful for manually parallelizing loops containing manually-ordered dependencies as described in Chapter 6, "Advanced shared memory programming."

### **Parallelizing loops with calls**

`loop_parallel` can be useful for manually parallelizing loops containing procedure calls. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
DO I = 1, N
    X(I) = FUNC(I)
ENDDO
```

The call to FUNC in this loop would normally prevent it from parallelizing. However, if you are sure that FUNC has no side effects (i.e. it does not modify its argument, it does not modify any COMMON variables, it performs no I/O, and it doesn't call any procedures that have side effects) and is compiled for reentrancy using the `-re` option (the default on SPP Series compilers), this loop can be safely parallelized as shown. If FUNC does have side effects or is not reentrant, this loop may yield wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_parallel
for (i=0; i<n; i++)
    x[i] = func(i);
```

## Note

In some cases, global register allocation can interfere with `loop_parallel` loops that contain procedure calls. Refer to the "Global register allocation" section of Chapter 3, "Compiler optimizations," for more information.

### Unparallelizable loops

The compiler will not parallelize any loop that does not have a number of iterations determinable prior to loop invocation at execution time, even when `loop_parallel` is specified. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
      DO WHILE (A(I) .GT. 0) !WON'T PARALLELIZE
      .
      .
      A(I) = ...
      .
      .
      ENDDO
```

There is no way the compiler can determine the loop's iteration count prior to loop invocation here, so the loop cannot be parallelized.

An analogous C example follows:

```
#pragma _CNX loop_parallel
while(a(i) > 0) { /* won't parallelize */
    .
    .
    a[i] = ...
    .
    .
}
```

---

## Task parallelization

The compiler does not automatically parallelize code outside a loop, but you can use tasking directives and pragmas to instruct the compiler to parallelize such code. The `begin_tasks` directive and pragma tells the compiler to begin parallelizing a series of tasks. The `next_task` directive and pragma marks the end of a task and the start of the next task. The `end_tasks` directive and pragma marks the end of a series of tasks to be parallelized and prevents execution from continuing until all tasks have completed. The sections of code delimited by these directives is referred to as a *task list*.

Within a task list, the compiler does not check for data dependencies, perform variable privatization, or perform parallelization analysis; you must synchronize any dependencies between tasks manually, manually privatize data as necessary, and if you wish to exploit nested parallelism, you must explicitly do so using directives or pragmas to define thread-parallel constructs within the node-parallel task list. In absence of a `nodes` or `threads` attribute, `begin_tasks` defaults to thread parallelization.

The Fortran tasking directives have the following forms:

```
C$DIR BEGIN_TASKS [ (attribute-list) ]
C$DIR NEXT_TASK
C$DIR END_TASKS
```

The C tasking pragmas have the following forms:

```
#pragma _CNX begin_tasks [ (attribute-list) ]
#pragma _CNX next_task
#pragma _CNX end_tasks
```

The optional *attribute-list* can contain one of the following attributes:

- `ordered`
- `nodes`
- `threads`
- `max_threads=m`
- `ordered, nodes`
- `ordered, threads`
- `ordered, max_threads=m`
- `nodes, max_threads=m`
- `threads, max_threads=m`
- `ordered, nodes, max_threads=m`
- `ordered, threads, max_threads=m`

The `ordered` attribute causes the tasks to be initiated in their lexical order; i.e. the first task in the sequence begins to run on its respective thread before the second and so on. In the absence of the `ordered` argument, the starting order will be indeterminate. Note that while this argument ensures an ordered starting sequence, it does not provide any synchronization between tasks, and does not guarantee any particular ending order. You can manually synchronize the tasks as described in Chapter 6, “Advanced shared memory programming,” if necessary.

The `nodes` attribute causes the tasks to run node-parallel, on one thread per available hypernode. If you want to exploit nested parallelism inside a `begin_tasks (nodes)` list, you must do so explicitly using thread-parallel constructs within the tasks.

The `threads` attribute causes the tasks to run thread-parallel, and is the default. As with parallel loops, node parallelism cannot be nested within thread parallelism in task lists.

The `ordered`, `nodes` and `ordered, threads` attributes cause the tasks to run ordered node-parallel and ordered thread-parallel, respectively.

The attributes specifying `max_threads = m` will run on no more than  $m$  threads, where  $m$  is an integer constant of known value at compile time. As shown, these attributes can include any combination of thread- or node-parallel, ordered or unordered execution.

---

## Caution

---

The compiler performs no dependency checking or synchronization on the code delimited by the tasking directives and pragmas; you must ensure that dependencies do not exist or insert your own synchronization code if necessary. Synchronization is discussed in Chapter 6, “Advanced shared memory programming.”

The following Fortran example shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel:

```
C$DIR BEGIN_TASKS
      parallel task 1
C$DIR NEXT_TASK
      parallel task 2
C$DIR NEXT_TASK
      parallel task 3
C$DIR END_TASKS
```

This example specifies thread-parallelism by default. The compiler transforms the above code into a parallel loop and creates machine code equivalent to the following Fortran:

```
C$DIR LOOP_PARALLEL (THREADS)
      DO 40 I = 1, 3
          GOTO (10, 20, 30) I
10     parallel task 1
          GOTO 40
20     parallel task 2
          GOTO 40
30     parallel task 3
          GOTO 40
40     CONTINUE
```

If there are more tasks than available threads, some threads will execute multiple tasks; if there are more threads than tasks, some threads will not execute tasks.

The `end_tasks` directive and `pragma` acts as a barrier; all parallel tasks must complete before the code following the `end_tasks` can execute.

### Examples

The following Fortran example illustrates how to use these directives to specify simple task-parallelism:

```
C$DIR BEGIN_TASKS
      DO I = 1, N - 1
          A(I) = A(I+1) + B(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X, Y)
C$DIR NEXT_TASK
      C(1:1000:2) = D(1:500)
C$DIR END_TASKS
```

In this example, one thread executes the `DO I` loop, another thread executes the `CALL TSUB(X, Y)`, and a third thread assigns the elements of the array `D` to every other element of `C`. These threads execute in parallel, but their starting and ending orders are indeterminate.

Unless the `NODES` attribute is supplied with the `BEGIN_TASKS` directive, the tasks are thread-parallelized. This means that there is no room for nested parallelization within the individual parallel tasks of this example, so the forward LCD on the `DO I` loop is inconsequential; there is no way for the loop to run but serially. The Fortran 90 array assignment in the last task will not parallelize either, even though it is technically parallelizable.

An analogous C example follows:

```
#pragma _CNX begin_tasks, task_private(i)
for(i=0;i<n-1;i++)
    a[i] = a[i+1] + b[i];
#pragma _CNX next_task
tsub(x,y);
#pragma _CNX next_task
for(i=1;i<=500;i++)
    c[i*2-1] = d[i];
#pragma _CNX end_tasks
```

The loop induction variable *i* must be manually privatized here because it is used to control loops in two different tasks. If *i* wasn't private, both tasks would modify it, causing wrong answers. This is not necessary in the Fortran example because the second loop is implemented as a Fortran 90 array assignment, for which the compiler generates an independent induction variable. The `task_private` directive and `pragma` is described in detail in the "task\_private" section of this chapter.

Nested task parallelism is also possible. In order to nest any parallelism on SPP Series machines, thread-parallelism must be nested within node-parallelism; when nesting tasking directives or `pragmas`, `begin_tasks(nodes)` must enclose `begin_tasks(threads)`. Also, if a node-parallel task contains a parallel loop, the loop cannot go node-parallel; if you want to parallelize it, you must explicitly do so using a thread-parallel construct. Thread-parallelism nested within node-parallelism can only run on the threads of the hypernode it is contained within.

The following Fortran example is more involved, and exploits two-dimensional parallelism:

```
C$DIR BEGIN_TASKS (NODES)
C$DIR LOOP_PARALLEL (THREADS)
  DO I = 1,N
    IF (B(I) .NE. 0) THEN
      A(I) = B(I)*C(I)
    ELSE
      A(I) = C(I)*D(I)
    ENDIF
  ENDDO
C$DIR NEXT_TASK
C$DIR BEGIN_TASKS (THREADS)
  CALL T1SUB()
C$DIR NEXT_TASK
  CALL T2SUB()
C$DIR NEXT_TASK
  CALL T3SUB()
C$DIR END_TASKS ! (THREADS)
C$DIR NEXT_TASK
  X(1:1000) = Y(1:1000)
C$DIR END_TASKS ! (NODES)
```

Here, the first node-parallel task contains a LOOP\_PARALLEL (THREADS) loop that goes parallel on the threads of the hypernode on which this task is running. The second node-parallel task contains a task list of three subroutine calls, each of which run on a separate thread within the hypernode. The third node-parallel task contains a Fortran 90 array section assignment which, while parallelizable, runs serially because automatic parallelization analysis is disabled within the task list.

An analogous C example follows:

```
#pragma _CNX begin_tasks(nodes)
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0;i<n;i++)
    if(b[i] != 0)
        a[i][j] = b[i]*c[i];
    else
        a[i] = c[i]*d[i];
#pragma _CNX next_task
    #pragma _CNX begin_tasks(threads)
    t1sub();
    #pragma _CNX next_task
    t2sub();
    #pragma _CNX next_task
    t3sub();
    #pragma _CNX end_tasks
#pragma _CNX next_task
for(j=0;j<1000;j++)
    x[j] = y[j];
#pragma _CNX end_tasks
```

Task parallelism can become even more involved, as described in Chapter 6, “Advanced shared memory programming.”

---

## Critical sections

The `critical_section` and `end_critical_section` directives and pragmas allow you to specify sections of code in parallel loops or tasks that must be executed by only one thread at a time. These directives cannot be used for ordered synchronization within a `loop_parallel(ordered)` loop, but are suitable for simple synchronization in any other `loop_parallel` loops.

A `critical_section` directive or pragma and its associated `end_critical_section` must appear in the same procedure, but they do not have to appear in the same procedure as the parallel construct in which they are used; i.e. the pair can appear in a procedure called from a parallel loop.

As discussed in this chapter, these directives have the following form in Fortran:

```
C$DIR CRITICAL_SECTION
C$DIR END_CRITICAL_SECTION
```

The C pragmas have the following form:

```
#pragma _CNX critical_section
#pragma _CNX end_critical_section
```

The `critical_section` directive and `pragma` can take an optional `gate` attribute which allows the declaration of multiple critical sections as described in Chapter 6, “Advanced shared memory programming;” however, we will only discuss simple critical sections here.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(FUNCTEMP)
      DO I = 1, N ! LOOP IS PARALLELIZABLE
          .
          .
          .
          FUNCTEMP = FUNC(X(I))
C$DIR  CRITICAL_SECTION
          SUM = SUM + FUNCTEMP
C$DIR  END_CRITICAL_SECTION
          .
          .
          .
      ENDDO
```

Here, since `FUNC` has no side effects and can be called in parallel, the `I` loop can be parallelized as long as the `SUM` variable is only updated by one thread at a time. The critical section created around `SUM` ensures this behavior.

Note that the `LOOP_PARALLEL` directive and the critical section are required to parallelize this loop because the call to `FUNC` would normally inhibit parallelization. If this call were not present, and if the loop didn't contain other parallelization inhibitors, the compiler would automatically parallelize the reduction of `SUM` as described in the “Reductions” section of Chapter 3, “Compiler optimizations”. However, the presence of the call necessitates the `LOOP_PARALLEL` directive, which prevents the compiler from automatically handling the reduction, and this, in turn, requires the critical section. Placing the call to `FUNC` outside of the critical section allows `FUNC` to be called in parallel, decreasing the amount of serial work within the critical section.

An analogous C example follows:

```
#pragma _CNX loop_parallel(ivar = i)
#pragma _CNX loop_private(functemp)
for(i=0;i<n;i++) {
    .
    .
    .
    functemp = func(x(i));
    #pragma _CNX critical_section
    sum = sum + functemp;
    #pragma _CNX end_critical_section
    .
    .
    .
}
```

In order to justify the cost of the compiler-generated synchronization code associated with the use of critical sections, loops which contain them must also contain a large amount of parallelizable (non-critical section) code. If you are unsure of the profitability of using a critical section to help parallelize a certain loop, time the loop with and without the critical section to see if parallelization justifies the overhead of the critical section.

---

### **-noautopar compiler option**

You can disable *automatic* loop parallelism by specifying the `-noautopar` option on the compiler command line. This option causes the compiler to parallelize only those loops which are immediately preceded by a `loop_parallel` or `prefer_parallel` directive or `pragma`; all other loops, even if they could normally be automatically parallelized, are not analyzed for parallelization. Since the compiler does not automatically find parallel tasks, user-specified task parallelization is not affected by this option. `-noautopar` is only meaningful when specified with the `-O3` option.

---

### **Reentrant compilation**

SPP Series Convex Fortran and C compilers both compile for reentrancy by default. Reentrant compilation causes procedures to store uninitialized local variables on the stack; no locals can carry values from one invocation of the procedure to the next (unless the variables appear in Fortran COMMON blocks or DATA or SAVE statements). This allows loops containing procedure calls to

be manually parallelized, assuming no other inhibitors of parallelization exist.

When procedures are called in parallel, each thread receives a private stack on which to allocate local variables. This allows each parallel copy of the procedure to manipulate its local variables without interfering with any other copy's locals of the same name. When the procedure returns and the parallel threads join, all values on the stack are lost.

Because the compiler cannot tell if a procedure is going to be called in parallel at runtime, it must create both parallel and serial copies—called *clones*—of every profitably parallelizable loop in the procedure. At runtime, the appropriate clone is executed based on whether the procedure is being called in parallel. This is similar to the dynamic selection optimization (discussed in Chapter 3, “Compiler optimizations”) but dynamic selection for reentrancy is not workload-based, and is not subject to any limit on the number of loop clones. Dynamic selection for reentrancy is unaffected by the compiler options, directives and pragmas used with workload-based dynamic selection.

Reentrant compilation can be disabled with the `-nore` option; keep in mind that calling a procedure that is compiled with this option in parallel will result in wrong answers.

---

## Default stack size

Thread 0's stack can grow to the size specified in the `maxssiz` operating system tunable. Refer to the *SPP-UX System Administration Guide* for more information on tunables.

Any threads your program spawns (as the result of `loop_parallel` or tasking directives or pragmas, for example) receive a default stack size of 8 Mbytes. This means that if a parallel construct declares more than 8 Mbytes of `loop_private` or `task_private` data, or if a subroutine with more than 8 Mbytes of local data is called in parallel, you must modify the stack size of the spawned threads via the `CPS_STACK_SIZE` environment variable. Under `csh`, this can be done with the following command:

```
setenv CPS_STACK_SIZE size_in_kbytes
```

Where *size\_in\_kbytes* is the desired stack size in kbytes. This value is read at program startup; it cannot be changed during execution.

---

## Loop- and task-specific data privatization

Once assigned, the memory classes mentioned in Chapter 1, "Introduction," and discussed in detail in Chapter 5, "Memory classes," are in effect throughout your entire program. Any loops that manipulate variables which have been explicitly assigned a memory class must be manually parallelized, and once a variable is assigned a class, its class cannot change. While very efficient programs can be written using these memory classes, they also require a great deal of manual intervention.

To get around these problems, the Convex SPP Series C and Fortran compilers support the `loop_private`, `task_private` and `parallel_private` directives and pragmas. The `save_last` directive and pragma is provided to save the value of `loop_private` data objects assigned in the last iteration of the loop. These directives and pragmas allow you to easily privatize parallel loop or task data temporarily; when used with `prefer_parallel`, they do so without inhibiting any automatic compiler optimizations. They can help you further increase the performance of your shared memory program with less extra work than is required when using the standard memory classes accompanying manual parallelization and synchronization.

You can use these directives on local variables and arrays of any type, but they should not be used on data assigned one of the static or dynamic memory classes (`thread_private`, `node_private`, `near_shared`, `far_shared` or `block_shared`).

In some cases, data declared `loop_private`, `task_private`, or `parallel_private` is stored on the stacks of the spawned threads. Spawned thread stacks default to 8 Mbytes in size; if the amount of `loop_private`, `task_private` or `parallel_private` data declared exceeds this, you can use the `CPS_STACK_SIZE` environment variable to increase the default. Refer to the "Default stack size" section of this chapter for more information.

---

### `loop_private`

The `loop_private` directive and pragma declares a list of variables and/or arrays private to the immediately following Fortran DO or C for loop. The compiler assumes that data objects declared to be `loop_private` have no loop-carried dependencies with respect to the parallel loops in which they are used. If dependencies exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, "Advanced shared memory programming."

loop\_private data must be local to the procedure in which it is declared loop\_private. loop\_private array dimensions must be determinable at compile-time.

Each parallel thread of execution receives a private copy of the loop\_private data object for the duration of the loop; no starting values can be assumed for the data, and unless a save\_last directive or pragma is specified (as described in a following section), no ending value can be assumed. If a loop\_private data object is referenced within an iteration of the loop, it must have been assigned a value previously on that same iteration.

In Fortran, the LOOP\_PRIVATE directive has the following form:

```
C$DIR LOOP_PRIVATE (namelist)
```

In C, the pragma has the following form:

```
#pragma _CNX loop_private(namelist)
```

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following loop. *namelist* cannot contain structures or dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
C$DIR LOOP_PRIVATE(S)
      DO I = 1, N
C      S IS ONLY CORRECTLY PRIVATE IF AT LEAST
C      ONE IF TEST PASSES ON EACH ITERATION:
          IF(A(I) .GT. 0) S = A(I)
          IF(U(I) .LT. V(I)) S = V(I)
          IF(X(I) .LE. Y(I)) S = Z(I)
          B(I) = S * C(I) + D(I)
      ENDDO
```

An apparent LCD on *S* exists in this example; if none of the IF tests are true on a given iteration, the value of *S* must wrap around from the previous iteration. The LOOP\_PRIVATE(*S*) directive indicates to the compiler that *S* does, in fact, get assigned on every iteration, and therefore it is safe to parallelize this loop.

Note that if on any iteration none of the IF tests pass, an actual LCD exists and privatizing *S* will result in wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_private(s)
for(i=0;i<=n;i++) {
/* s is only private if at least one if
   test passes: */
   if(a[i] > 0) s = a[i];
   if(u[i] < v[i]) s = v[i];
   if(x[i] < y[i]) s = z[i];
   b[i] = s * c[i] + d[i];
}
```

### Using loop\_private with loop\_parallel

Since the compiler does not automatically perform variable privatization in loop\_parallel loops, you must manually privatize loop data requiring privatization. This can be easily done using the loop\_private directive or pragma.

Consider the following Fortran example:

```
      SUBROUTINE PRIV(X,Y,Z)
      REAL X(1000), Y(4,1000), Z(1000)
      REAL XMFIED(1000)
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(XMFIED, J)
      DO I = 1, 4
C   INITIALIZE XMFIED; MFY MUST NOT WRITE TO X:
          CALL MFY(X, XMFIED)
          DO J = 1, 999
              IF (XMFIED(J) .GE. Y(I,J)) THEN
                  Y(I,J) = XMFIED(J) * Z(J)
              ELSE
                  XMFIED(J+1) = XMFIED(J)
              ENDIF
          ENDDO
      ENDDO
      END
```

Here, the LOOP\_PARALLEL directive is required to parallelize the I loop because of the call to MFY. The X and Y arrays are in shared memory by default. X and Z are not written to, and the portions of Y written to in the J loop's IF statement are disjoint, so these shared arrays require no special attention. The local array XMFIED, however, is written to. But because XMFIED carries no values into or out of the I loop, it can be privatized using LOOP\_PRIVATE. This gives each thread running the I loop its own private copy of XMFIED, eliminating the expensive necessity of synchronized access to XMFIED. Note that a loop carried dependency exists for XMFIED in the J loop, but since this loop runs serially on each processor, this dependency is safe.

J is privatized as discussed in the "Privatizing induction variables in nested loops" section.

An analogous C example follows:

```
void priv(float x[1000], float y[4][1000], float z[1000]) {
    float xmfied[1000];
    int i,j;
#pragma _CNX loop_parallel(ivar=i), loop_private(xmfied,j)
    for(i=0;i<4;i++) {
        mfy(x,xmfied);
        for(j=0;j<999;j++) {
            if(xmfied[j] >= y[i][j]) y[i][j] = xmfied[j]*z[j];
            else xmfied[j+1] = xmfied[j];
        }
    }
}
```

### Denoting induction variables in parallel loops

To safely parallelize a loop with the `loop_parallel` directive or `pragma`, the compiler must be able to correctly determine the loop's primary induction variable.

The compiler can find primary Fortran DO loop induction variables; it may, however, have trouble with DO WHILE or hand-rolled Fortran loops, and with all `loop_parallel` loops in C. Therefore, when you use the `loop_parallel` directive or `pragma` to manually parallelize a loop other than an explicit Fortran DO loop, you should indicate the loop's primary induction variable using the `IVAR=indvar` attribute to `loop_parallel`. Consider the following Fortran example:

```
      I = 1
C$DIR LOOP_PARALLEL(IVAR = I)
10    A(I) = ...
      .
      .          ! ASSUME NO DEPENDENCIES
      .
      I = I + 1
      IF(I .LE. N) GOTO 10
```

This is a hand-rolled loop that uses I as its primary induction variable. To ensure parallelization, the `LOOP_PARALLEL` directive has been placed immediately before the start of the loop, and the induction variable, I, has been specified.

Primary induction variables in C loops can be difficult for the compiler to find, so `ivar` is required in all `loop_parallel` C loops. Its use is shown in the following example:

```
#pragma _CNX loop_parallel(ivar = i)
  for(i=0; i<n; i++) {
    a[i] = ...;
    .
    . /* assume no dependencies */
    .
  }
}
```

Secondary induction variables are variables used to track loop iterations even though they do not appear in the Fortran DO statement. They cannot appear in addition to the primary induction variable in the C `for` statement. Such variables *must* be a function of the primary loop induction variable; they cannot be independent. Secondary induction variables must also either be assigned a memory class manually (as described in Chapter 5, “Memory classes”) or declared `loop_private`.

The following Fortran example contains an incorrectly incremented secondary induction variable:

```
C WARNING: INCORRECT EXAMPLE!!!!
      J = 1
C$DIR LOOP_PARALLEL
      DO I = 1, N
          J = J + 2 ! WRONG!!!
```

Here, `J` is *not* a legal secondary induction variable because it is not a function of `I`. It is not private because the value assigned to it in the current iteration is a function of its value in the previous iteration. This example can be corrected by privatizing `J` and making it a function of `I`, as shown below.

```
C CORRECT EXAMPLE:
      J = 1
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(J) ! J IS PRIVATE
      DO I = 1, N
          J = (2*I)+1 ! J IS PRIVATE
```

Here `J` is a legal secondary induction variable because it is a function of `I`, and can be safely privatized.

In C, secondary induction variables are sometimes included in for statements, as shown in the following example:

```
/* warning: unparallelizable code follows */
#pragma _CNX loop_parallel(ivar = i)
for(i=j=0; i<n;i++,j+=2) {
    a[i] = ...;
    .
    .
    .
}
}
```

Because secondary induction variables must be private to the loop and must be a function of the primary induction variable, this example cannot be safely parallelized using `loop_parallel(ivar = i)`. In the presence of this directive, the secondary induction variable will not be recognized. To manually parallelize this loop, you must remove `j` from the `for` statement and either privatize it and make it a function of `i`, or declare `j` to be shared (which is the default storage class) and increment it within a critical section inside the loop. This latter method is costly in terms of synchronization overhead and may degrade the performance of the loop.

The following example demonstrates how to restructure the loop so that `j` is a valid secondary induction variable:

```
#pragma _CNX loop_parallel(ivar = i)
#pragma _CNX loop_private(j)
for(i=0; i<n; i++) {
    j = 2*i;
    a[i] = ...;
    .
    .
    .
}
}
```

This method runs faster than placing `j` in a critical section because it requires no synchronization overhead, and the private copy of `j` used here can typically be more quickly accessed than a shared variable.

### Privatizing induction variables in nested loops

The induction variables of nonparallel loops that are contained within parallel loops must be declared `loop_private` with respect to their closest enclosing parallel loop.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL (THREADS)
C$DIR LOOP_PRIVATE (J)
  DO I = 1, N      ! I LOOP GOES PARALLEL
    DO J = 1, M    ! J LOOP IS SERIAL
      .
      .
      .
    ENDDO
  ENDDO
```

Here, the `LOOP_PARALLEL` causes the `I` loop to be parallelized across threads. The `J` loop, then, runs serially. `J` must be private with respect to the `I` loop so that the threads that run the `I` loop do not attempt to update the same copy of `J`. If the loop is automatically parallelized by the compiler, or parallelized due to the presence of a `PREFER_PARALLEL` directive, this privatization will be automatic. But the presence of the `LOOP_PARALLEL` directive requires manual privatization.

An analogous C example follows:

```
#pragma _CNX loop_parallel(threads, ivar = i)
#pragma _CNX loop_private(j)
for(i=0;i<50;i++) {
  for(j=0;j<50;j++) {
    .
    .
    .
  }
}
```

This also applies to nested parallel outer loops. In this case loop variables contained within a parallel construct—even if they are used in a parallel loop themselves—must be declared private with respect to the innermost enclosing parallel loop. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL (NODES), LOOP_PRIVATE (J)
  DO I = 1, N      ! I LOOP GOES NODE PAR
    C$DIR LOOP_PARALLEL (THREADS)
    C$DIR LOOP_PRIVATE (K)
      DO J = 1, M    ! J LOOP GOES THREAD PAR
        DO K = 1, L ! K LOOP IS SERIAL
          .
          .
          .
        ENDDO
      ENDDO
    ENDDO
```

Here, LOOP\_PARALLEL is used to parallelize the I loop across hypernodes, and the J loop across processors on each hypernode. K must be declared private to the J loop to ensure that the thread-parallel threads don't interfere with each other in updating it. J must be declared private to the I loop to insure that each node-parallel thread gets its own copy.

An analogous C example follows:

```
#pragma _CNX loop_parallel(nodes, ivar=i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar=j)
#pragma _CNX loop_private(k)
    for(j=0;j<m;j++) {
        for(k=0;k<l;k++) {
            .
            .
            .
        }
    }
}
```

---

### **task\_private**

The `task_private` directive declares a list of variables and/or arrays private to the immediately following tasks; it serves the same purpose for parallel tasks that `loop_private` serves for loops.

The `task_private` directive must immediately precede or appear on the same line as its corresponding `begin_tasks` directive. The compiler assumes that data objects declared to be `task_private` have no dependencies between the tasks in which they are used. If dependencies exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, "Advanced shared memory programming."

Each parallel thread of execution receives a private copy of the `task_private` data object for the duration of the tasks; no starting or ending values can be assumed for the data. If a `task_private` data object is referenced within a task, it must have been assigned a value previously in that task.

In Fortran, the `TASK_PRIVATE` directive has the following form:

```
C$DIR TASK_PRIVATE (namelist)
```

In C, the pragma has the following form:

```
#pragma _CNX task_private(namelist)
```

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following tasks. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
REAL*8 A(1000), B(1000), WRK(1000)
.
.
.
C$DIR BEGIN_TASKS, TASK_PRIVATE(WRK)
DO I = 1, N
  WRK(I) = A(I)
ENDDO
DO I = 1, N
  A(I) = WRK(N+1-I)
.
.
.
ENDDO
C$DIR NEXT_TASK
DO J = 1, M
  WRK(J) = B(J)
ENDDO
DO J = 1, M
  B(J) = WRK(M+1-J)
.
.
.
ENDDO
C$DIR END_TASKS
```

Here, the *WRK* array is used in the first task to temporarily hold the *A* array so that its order can be reversed. It serves the same purpose for the *B* array in the second task. *WRK* is assigned before it is used in each task.

An analogous C example follows:

```
float a[1000], b[1000], wrk[1000];
.
.
.
#pragma _CNX task_private(wrk)
#pragma _CNX begin_tasks(nodes)
for(i=0;i<n;i++)
    wrk[i] = a[i];
for(i=0;i<n;i++) {
    a[i] = wrk[n-1-i];
    .
    .
    .
}
#pragma _CNX next_task
for(j=0;j<m;j++)
    wrk[j] = b[j];
for(j=0;j<m;j++) {
    b[j] = wrk[m-1-j];
    .
    .
    .
}
#pragma _CNX end_tasks
```

---

### **save\_last**

The `save_last` directive and `pragma` allow you to save the final value of all `loop_private` data objects assigned in the last iteration of the immediately following loop. The values must be assigned in the last iteration; if the assignment is executed conditionally, it is the programmer's responsibility to ensure that the condition is met and the assignment executes. Incorrect answers can result if the assignment does not execute on the last iteration. For `loop_private` arrays, only those elements of the array assigned on the last iteration will be saved.

In Fortran, the `SAVE_LAST` directive has the following form:

```
C$DIR SAVE_LAST
```

In C, the `pragma` has the following form:

```
#pragma _CNX save_last
```

`save_last` must appear immediately before or after the associated `loop_private` directive or `pragma`, or on the same line.

Consider the following Fortran example:

```
C$DIR LOOP_PRIVATE(ATEMP), SAVE_LAST
DO I = 1, N
  IF (I .EQ. D(I)) ATEMP = A(I)
  IF (I .EQ. E(I)) ATEMP = B(I)
  IF (I .EQ. F(I)) ATEMP = C(I)
  A(I) = B(I) + C(I)
  B(I) = ATEMP
ENDDO
.
.
.
IF(ATEMP .GT. AMAX) THEN
.
.
.
```

Here, the `LOOP_PRIVATE` variable `ATEMP` is conditionally assigned in the loop; in order for `ATEMP` to be truly private, the programmer must be sure that at least one of the conditions is met so that `ATEMP` is assigned on every iteration. When the loop terminates, the `SAVE_LAST` directive ensures that `ATEMP` contains the value it is assigned on the last iteration, which is used later in the program.

An analogous C example follows:

```
#pragma _CNX loop_private(atemp), save_last
for(i=0;i<n;i++) {
  if(i==d[i]) atemp = a[i];
  if(i==e[i]) atemp = b[i];
  if(i==f[i]) atemp = c[i];
  a[i] = b[i] + c[i];
  b[i] = atemp;
}
.
.
.
if(atemp > amax) {
.
.
.
```

Note that the `save_last` directive can be misleading in certain loop contexts. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(S), SAVE_LAST
DO I = 1, N
  IF (G(I) .GT. 0) THEN
    S = G(I) * G(I)
  ENDIF
ENDDO
```

While it may appear that the last value of `S` assigned (on whatever iteration) is saved in this example, you must remember that the `SAVE_LAST` directive applies only to the last (Nth) iteration, without regard for any conditionals contained in the loop. For `SAVE_LAST` to be valid here, `G(N)` must be greater than 0 so that the assignment to `S` takes place on the final iteration. Obviously, if this condition can be predicted, the loop can be more efficiently written to exclude the `IF` test, so the presence of a `SAVE_LAST` in such a loop is suspect.

A `save_last` directive or `pragma` causes the compiler to peel the last iteration of the loop to facilitate saving. In this way, all but the last iteration of the loop can be parallelized; the final iteration runs serially, and carries out the final assignment to the saved variable.

The optimization report describes such peeled variables in the loop table, the test table, and the privatization table. Refer to Appendix C for more information on the optimization report.

---

## Performance analysis

If you are willing to make the manual modifications to your program suggested here and in Chapter 6, “Advanced shared memory programming,” you may wish to use `CXpa`, the Convex visual profiler, to analyze its performance and determine the areas of code that would most benefit from manual optimizations. `CXpa` has both an interactive graphical user interface and a character-based line mode. Through either interface, it profiles the following performance factors on a per-thread basis over user-selectable regions of application code without source modification.

- On all SPP Series architectures:
  - Wall clock time
  - CPU time
  - Concurrency factor (CPU/Wall clock time)
  - Iteration/execution counts
- On SPP 1000 systems:
  - Locally resolved (CTIring not used) cache miss counts and latency

- Remotely resolved (CTIring used) cache miss counts and latency
- Locally and remotely resolved (total) cache miss counts and latency
- Average cache miss latency
- On SPP 1200 systems:
  - Data cache misses, accesses, and latency
  - Data cache hit rates
  - Average data cache miss latency
  - Instruction cache misses and latency
  - Average instruction cache miss latency
  - Instructions completed
  - Clock cycles
  - Average clock cycles per instruction

After profiling, CXpa supports visualization of profiling results in 2D and 3D graphs. CXpa is an optional product. Refer to the *CXpa Reference for Exemplar Systems* (DSW-605) or contact your Convex sales representative for more information.



Chapter 2, “Architecture overview,” discusses the three partitions of physical memory available on SPP Series systems. These partitions are

- Hypernode-local
- Subcomplex-global
- CTIcache.

Hypernode-local memory is accessed via the `thread_private` and `node_private` virtual memory classes. Subcomplex-global memory is accessed via the `near_shared`, `far_shared` and `block_shared` virtual memory classes. CTIcache physical memory holds copies of shared memory data which is not resident in the hypernode’s physical memory, but is accessed by threads running on the hypernode.

---

## Caution

---

The memory classes discussed here are only of interest to programmers who wish to manually optimize their shared memory programs by using compiler directives or pragmas to partition memory and otherwise control compiler optimizations as discussed in Chapter 6, “Advanced shared memory programming.” *Using these memory classes requires you to manually handle parallelization.*

---

## Private versus share memory

Private and shared data are differentiated by their accessibility, and, as noted above, by the physical memory classes in which they are stored.

Both `node_private` and `thread_private` data are stored in hypernode-local memory, and are therefore inaccessible to any hypernode other than the one on which they reside (in the case of `thread_private`, access is further restricted to the declaring thread). Latency is identical for private data items that must be fetched from main memory. `near_shared`, `far_shared` and `block_shared` data, on the other hand, are stored in subcomplex-global physical memory and are therefore accessible from any hypernode in the subcomplex on which the process is running. Main memory latency can vary for the shared memory classes depending on whether or not the data is resident on the requesting hypernode.

---

## Memory class addressing

Figure 20 shows the virtual addresses associated with a data item stored in each memory class by a single process running on a conceptual 4-hypernode, 8-processor-per-hypernode system. Each oval represents a unique virtual address for the same data item within a class; the memory class is indicated by the oval's fill pattern as explained in the illustration.

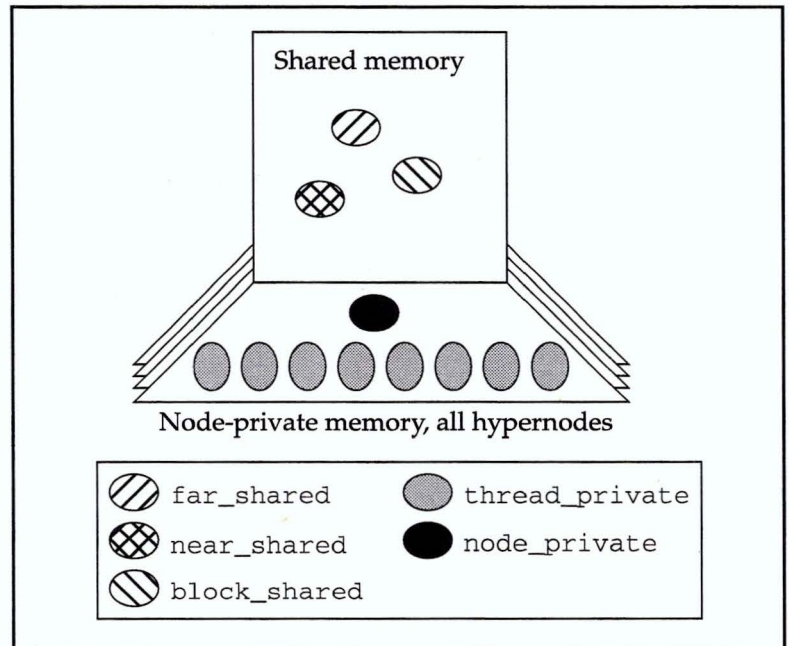
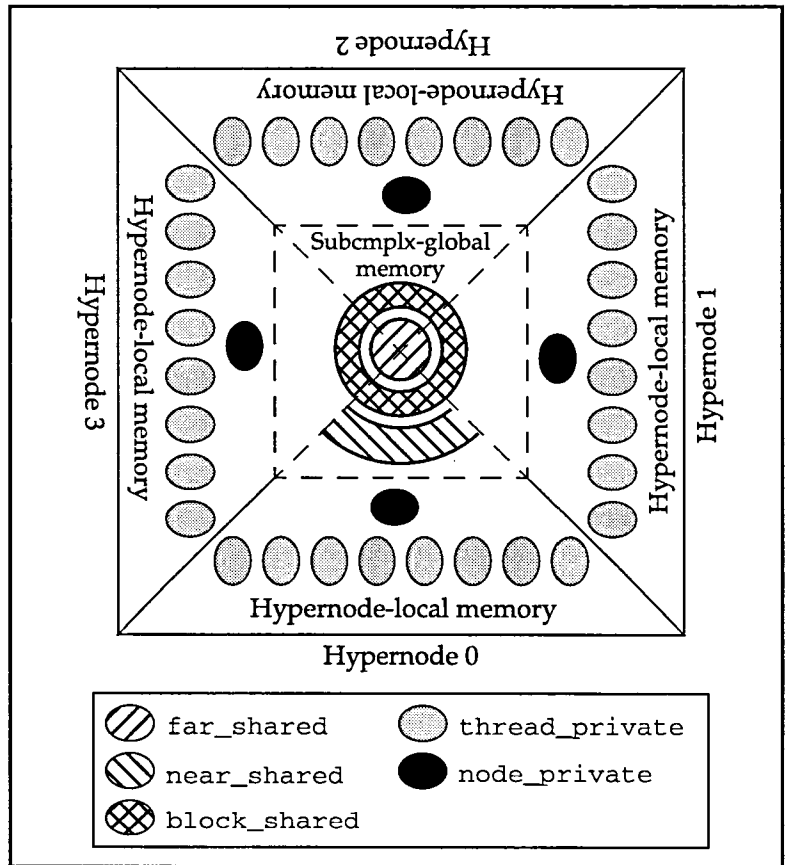


Figure 20 Virtual addresses for various memory classes

As shown, the shared data items are accessible from any hypernode using the same virtual address; the `node_private` data item has a single unique virtual address; the `thread_private` item has up to 8 unique virtual addresses, one for each thread within a hypernode.

Figure 21 shows the physical addresses associated with the data items shown in Figure 20, for an identical SPP Series system. For illustrative purposes, all shared data in is assumed to be array data, and is represented by circles and portions of circles rather than ovals.



**Figure 21** Physical addresses for various memory classes

Figure 21 shows the physical memory replication of the private memory classes, as well as the distributed nature of the `far_shared` and `block_shared` classes. You can control hypernode placement of the `near_shared` class, shown here residing physically on hypernode 0. Note that all hypernodes can access this data, since it resides in subcomplex-global physical memory.

The subsections that follow explain virtual and physical addressing for each memory class in detail.

---

### **thread\_private**

`thread_private` data is private to each thread of a process. Each `thread_private` data object has its own unique virtual address within a hypernode. For statically-declared `thread_private` data on multihypernodal subcomplexes, these unique virtual addresses are replicated on each hypernode. If the data is dynamically allocated, no virtual address replication is done; all virtual addresses are unique.

These virtual addresses map to unique physical addresses in hypernode-local physical memory on each hypernode; therefore, while a `thread_private` data item can have identical virtual addresses on different hypernodes, these virtual addresses map to unique physical addresses. For example, on a 4-hypernode, 8-processor-per-hypernode system, a single, statically allocated `thread_private` data item is accessed by 8 virtual addresses that map to 32 physical addresses. On a 7-hypernode, 8-processor-per-hypernode system, the same data item is accessed by 8 virtual addresses which map to 56 physical addresses.

Obviously, this physical address replication can cause a single data item to occupy a large amount of memory; similarly, virtual address replication subtracts from the total 4 Gbyte virtual address space available to the process.

Any sharing of `thread_private` data items between threads (regardless of whether they are running on the same hypernode) must be done by synchronized copying of the item into a shared variable, or by message passing.

`thread_private` data cannot be initialized in Fortran DATA statements.

---

### **node\_private**

`node_private` data is private to the threads running on a given hypernode. `node_private` data items have one virtual address, and any thread on a hypernode can access that hypernode's `node_private` data using the same virtual address. This virtual address maps to a unique physical address in hypernode-local memory. On a multihypernodal subcomplex, a physical copy of the data item is contained in each hypernode's hypernode-local memory, and this copy is accessed by the same virtual address on any hypernode.

---

This physical replication will multiply the amount of physical memory a `node_private` data item takes up by the number of hypernodes in the subcomplex.

Any sharing of `node_private` data items between hypernodes must be done by synchronized copying into a shared variable, or by message passing.

---

### **near\_shared**

`near_shared` data is accessible by any thread running on any hypernode of a subcomplex, but is physically stored entirely within the subcomplex-global memory of a particular hypernode, allowing faster access to the threads running on that hypernode. `near_shared` data has a single virtual address through which it is accessed by every thread. The data's physical placement in hypernode-global memory eliminates the need for replication. Threads running on the hypernode on which the data is stored can access it via the crossbar within 60 processor clocks; threads running on other hypernodes must access the data via their CTIcache if it is encached there; if not, they must fetch it over the SCI rings. These accesses can take over 200 processor clocks, depending on whether any coherency or bank conflicts are encountered.

As explained in the "Static assignments" section later in this chapter, the `near_shared` class is typically used for data which is accessed heavily by threads running on the hypernode on which it resides, but which must also be easily accessible to threads running on other hypernodes.

---

### **far\_shared**

`far_shared` data is accessible by any thread running on any hypernode, and, since it is distributed evenly across the subcomplex-global memory of all hypernodes in a subcomplex, it provides the best average access time when it is used equally by threads running on all available hypernodes. A `far_shared` data item has a single virtual address and a single physical address, but the virtual pages of `far_shared` data are mapped round-robin to physical memory pages on all the hypernodes in the subcomplex. The page size is 4KB.

As explained in the "Static assignments" section later in this chapter, the `far_shared` class is typically used for data which is accessed equally by all the hypernodes in a subcomplex.

The `far_shared` memory class is the default for any data not specifically classified by the programmer.

---

### **block\_shared**

A `block_shared` data item has a unique virtual and a unique physical address, and can be accessed by any thread running on any hypernode in the subcomplex. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the subcomplex-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a `block_shared` array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division.

`block_shared` allocation is only useful when combined with manual parallelization of loops that use the arrays. In this case it allows you to parallelize the loops such that a parallel section running on a given hypernode can access the portion of the array residing on that hypernode with low latency, but interhypernode access of the remaining elements is also possible.

Using the `block_shared` class is explained in detail in the "Dynamic assignments" section later in this chapter.

---

## **Memory class assignments**

In Fortran, compiler directives are used to assign memory classes to data items. In C, memory classes are assigned through the use of syntax extensions to Convex C, which are defined in header file `/usr/convex/all/include/spp_prog_model.h`. This file must be included in any C program which uses memory classes.

The general form for Fortran memory class directives is

```
C$DIR class_name (namelist)
```

where *namelist* is a comma-delimited list of variables, arrays, and/or common block names to be assigned the class *class\_name*. Common block names must be enclosed in slashes, and only entire common blocks can be assigned a class. This means that arrays and variables in *namelist* must not also appear in a common block, and must not be equivalenced to data objects in common blocks.

*class\_name* can be `THREAD_PRIVATE`, `NODE_PRIVATE`, `NEAR_SHARED`, `FAR_SHARED`, or `BLOCK_SHARED`. For `BLOCK_SHARED`, *namelist* must include only allocatable arrays.

These Fortran memory class declarations must appear with other specification statements; they cannot appear within executable statements.

In C, Convex type-qualifier extensions are used, so memory classes are assigned in variable declarations. The general form for assigning memory classes in C is

```
#include <spp_prog_model.h>
.
.
.
[storage_class_specifier] class_name type_specifier namelist
```

where

*storage\_class\_specifier*

Specifies a nonautomatic storage class

*class\_name*

Is the desired memory class (`thread_private`, `node_private`, `near_shared`, or `far_shared`; the `block_shared` class must be allocated dynamically, as described in the “`block_shared`” section on page 169)

*type\_specifier*

Is a standard C data type (`int`, `float`, etc.), and *namelist* is a comma-delimited list of variables and/or arrays of type *type\_specifier*.

In C, data objects which are assigned a memory class must have static storage duration. This means that if the object is declared within a function, it must have the storage class `extern` or `static`. If such an object is not given one of these storage classes, its storage class defaults to `automatic` and it is allocated on the stack. Stack-based objects cannot be assigned a memory class, and attempting to do so will result in a compile-time error.

Data objects declared at file scope and assigned a memory class need not specify a storage class. For more information on C scoping rules, refer to *The C Programming Language* (DSW-046).

## Note

All C code examples presented in this chapter assume that the line `#include <spp_prog_model.h>` appears above the C code presented. This header file maps user symbols to the implementation reserved space.

If you assign a memory class to a C structure, all structure members must be of the same class.

In both C and Fortran, once a data item is assigned a memory class, the class cannot be changed.

The Convex SPP Series compilers provide mechanisms for assigning memory classes statically and dynamically. Static assignments make sense for the private classes and for `far_shared` memory; dynamic assignments make most sense for `near_shared` and `block_shared` memory. The sections that follow explain both static and dynamic memory class assignments in detail.

---

## Static assignments

Static memory class assignments are physically located with variable type declarations in the source. Static memory classes are typically used with data objects that are accessed with equal frequency by all threads; these include objects of the `thread_private`, `node_private`, and `far_shared` classes. Static assignments for all classes are explained in the subsections that follow.

### `thread_private`

Since `thread_private` variables are replicated for every thread on every hypernode, static declarations make the most sense for them.

In Fortran, the `thread_private` memory class is assigned using the `THREAD_PRIVATE` compiler directive, as shown in the following example:

```
REAL*8 TPX(1000)
REAL*8 TPY(1000)
REAL*8 TPZ(1000), X, Y
COMMON /BLK1/ TPZ, X, Y
C$DIR THREAD_PRIVATE(TPX, TPY, /BLK1/)
```

Each array declared here is 8000 bytes in size, and each variable is 8 bytes, for a total of 24,016 bytes of data. The entire common block `BLK1` is placed in `thread_private` memory along with `TPX` and `TPY`. All memory space is replicated for each thread in hypernode-local physical memory on every hypernode.

`thread_private` variables and arrays cannot be initialized in Fortran `DATA` statements.

The following C example demonstrates several ways to declare `thread_private` storage in C. Note that the data objects

declared here are not scoped analogously to those declared in the Fortran example:

```
/* tpa is global: */
thread_private double tpa[1000];
func() {
    /* tpb is local to func: */
    static thread_private double tpb[1000];
    /* tpc, a and b are declared elsewhere: */
    extern thread_private double tpc[1000], a, b;
    .
    .
    .
}
```

C's `double` data type provides the same precision as Fortran's `REAL*8`. The `thread_private` data declared here occupies the same amount of memory as that declared in the Fortran example. `tpa` is available to all functions lexically following it in the file. `tpb` is local to `func` and inaccessible to other functions. `tpc`, `a`, and `b` are declared at filescope in another file that is linked with this one.

Assume a Fortran or C program containing the appropriate example is running on a 4-hypernode subcomplex with 8 processors per hypernode. Each data item will require 8 virtual addresses, for a total of 192,128 bytes of virtual space. These virtual addresses will map to 8 physical addresses per hypernode, or 32 total physical addresses per data item, requiring a total of 768,512 (32×24016) bytes of physical memory.

### **node\_private**

Since `node_private` variables are physically replicated on every hypernode, static declarations make the most sense for them.

In Fortran, the `node_private` memory class is assigned using the `NODE_PRIVATE` compiler directive, as shown in the following example:

```
REAL*8 XNP(1000)
REAL*8 YNP(1000)
REAL*8 ZNP(1000), X, Y
COMMON /BLK1/ ZNP, X, Y
C$DIR NODE_PRIVATE(XNP, YNP, /BLK1/)
```

Again, the data requires 24,016 bytes. The contents of `BLK1` are placed in `node_private` memory along with `XNP` and `YNP`. Space for each data item is replicated once per hypernode in hypernode-local physical memory. The same virtual address is used by each thread to access its hypernode's copy of a data item.

`node_private` variables and arrays can be initialized in Fortran DATA statements.

The following example shows several ways to declare `node_private` data objects in C.

```
/* npa is global: */
node_private double npa[1000];
func() {
    /* npb is local to func: */
    static node_private double npb[1000];
    /* npc, a and b are declared elsewhere: */
    extern node_private double npc[1000], a, b;
    .
    .
    .
}
```

The `node_private` data declared here occupies the same amount of memory as that declared in the Fortran example. Scoping rules for this data are similar to those given for the `thread_private` C example.

For either language example, assuming an 8 processor per hypernode, 4 hypernode system, each data item would require a single virtual address, for a total of 24,016 bytes of virtual space. These virtual addresses map to 4 physical addresses each, one per hypernode, for a total of 96,064 bytes of physical memory.

Because `node_private` data is physically replicated across hypernodes but not replicated in virtual memory, on multihypernodal subcomplexes it can effectively expand the physical space available to a process. For example, if a process declares 2 Gbytes of data `node_private`, the virtual addresses it uses to access this data map to a total of 16 Gbytes of physical memory on an 8 hypernode subcomplex (2 Gbytes per hypernode). Assuming this `node_private` data is made up of an array, you could manually split up a loop that manipulates the array to run on several hypernodes. Each hypernode would then compute its entire 2 Gbytes of the array; to the hypernode, this private copy appears to be the entire array, when in fact other hypernodes are working on private 2 Gbyte chunks with identical array names (and thus identical virtual addresses) that also appear to be the entire array. Through careful manual synchronization, the results of these hypernode-private computations can be shared through the use of “communication” arrays of the `far_shared` or `block_shared` memory class.

Such an approach approximates message passing using shared memory constructs, and can be beneficial when arrays contain quantities of data that surpass available virtual memory.

## near\_shared

Static assignments of the `near_shared` memory class are of limited usefulness, because they place the declared `near_shared` memory on hypernode 0. The purpose of declaring the `near_shared` memory is defeated unless the code that most frequently accesses this data happens to be running on hypernode 0, which is unlikely on a multihypernodal subcomplex.

However, a mechanism is provided for statically declaring `near_shared` memory. In Fortran, the `near_shared` memory class is statically assigned using the `NEAR_SHARED` compiler directive, as in the following example:

```
SUBROUTINE FUNC ()
  REAL*8 XNS(1000, 1000)
C$DIR NEAR_SHARED(XNS)
  .
  .
  .
```

Here, `XNS` is local to `FUNC()`.

`near_shared` variables and arrays can be initialized in Fortran `DATA` statements.

A similar example in C follows:

```
func() {
  static near_shared double xns[1000][1000];
  .
  .
  .
```

Here, `xns` is local to `func()`. Global declarations, and local declarations of the `extern` storage class, are also legal.

Both language examples allocate 8,000,000 bytes of virtual address space, which maps to 8,000,000 bytes of physical memory on hypernode 0. Any thread running on any other hypernode will experience interhypernode access latency when accessing this data. Therefore, this code only makes sense for single-hypernode systems, or for code that will always be executed on hypernode 0 only.

The true power of `near_shared` memory is realized only when it is resident on the hypernode which most frequently accesses it. The `near_shared` class is therefore most efficiently assigned dynamically at runtime, by the thread that will access the `near_shared` data object most often, on the hypernode on which that particular thread is running. Such dynamic allocation is

discussed in the “Dynamic assignments” section later in this chapter.

### **far\_shared**

Since `far_shared` memory is physically distributed among all hypernodes and is best used when all hypernodes will be accessing it with similar frequency, static declarations make the most sense.

In Fortran, the `far_shared` memory class is assigned as shown in the following example:

```
        SUBROUTINE FUNC()  
          REAL*8 XFS(1000,1000)  
C$DIR FAR_SHARED(XFS)  
        .  
        .  
        .
```

Here, `XFS` is local to `FUNC()`.

`far_shared` variables and arrays can be initialized in Fortran `DATA` statements.

A similar C example follows:

```
void func() {  
    static far_shared double xfs[1000][1000];  
    .  
    .  
    .  
}
```

Here, `xfs` is local to `func()`. Global declarations, and local declarations of the extern storage class, are also legal.

These declarations allocate 8,000,000 bytes of virtual address space, which is mapped to physical memory by 4KB pages round-robin to each hypernode on a multihypernodal subcomplex, beginning at hypernode 0. When all hypernodes are accessing the `far_shared` data with relatively equal frequency, this provides the best average access times. However, if your program only spawns threads on a subset of the hypernodes in a multi-hypernodal subcomplex, `far_shared` memory is a poor choice since it will still be distributed across all hypernodes in the subcomplex. If you know that your program will only spawn threads on one hypernode, use the `near_shared` class to allocate memory on that hypernode; if your program will spawn threads on multiple hypernodes but not every hypernode in the subcomplex, use the `block_shared` class as described in the “Dynamic assignments” section.

## **block\_shared**

The `block_shared` memory class can only be allocated dynamically, as described in the following section.

---

## **Dynamic assignments**

Dynamic memory class assignments are used with Convex Fortran ALLOCATABLE arrays and with the `memory_class_malloc` function in Convex C. The class assignments are located with variable declarations. As with static assignments, in Fortran, compiler directives are used to specify the desired memory class for a previously-declared data object; in C, the memory class is specified in the declaration using a type-qualifier extension. The allocation is done at the specific point in the program where the memory is needed, using the Fortran ALLOCATE statement or the C `memory_class_malloc` function. At this point, virtual memory is allocated, and the program's available virtual space is decreased by the amount of memory allocated. This virtual memory does not map to physical memory until the allocated data objects are referenced.

While any memory class can be dynamically assigned, the `block_shared` class can only be assigned dynamically, and the `near_shared` class is most useful when dynamically assigned.

## **Memory class pointers**

All shared memory classes are accessible to all threads when dynamically allocated in serial code, regardless of the allocating thread, because all threads access these classes from the same physical memory using the same virtual addresses. However, in Fortran, if more than one thread in a parallel construct attempts to allocate a shared class array, only the last allocation will exist. This is because there can only be one (internal) pointer to the allocated array; by default, this pointer is of the same class as the allocated memory, and each allocating thread resets this shared pointer. This problem is overcome by adding class-specification directives of the following form:

```
C$DIR class_name_POINTER (allocatable-namelist)
```

where *class\_name* is one of `THREAD_PRIVATE`, `NODE_PRIVATE`, `NEAR_SHARED`, or `FAR_SHARED` and *allocatable-namelist* is a comma-delimited list of arrays previously declared to be ALLOCATABLE in the same procedure. *class\_name* cannot contain `BLOCK_SHARED` because the `BLOCK_SHARED` class is specifically designed to hold array objects, and a pointer is a scalar object. The private classes are included in *allocatable-namelist* because it is

often only necessary to access a particular shared array from the particular thread or hypernode on which the array is being manipulated.

Allocatable private arrays are only accessible from the thread which allocates them; threads executing `ALLOCATE` statements in parallel will each be able to access the private array they allocate. Private arrays allocated outside of parallel constructs will only be accessible by thread 0.

The C `memory_class_malloc` function is very similar to standard `malloc` and has the following form:

```
memcls_ptr = void *memory_class_malloc(size_t bytes, int class_name);
```

where *memcls\_ptr* is a previously-declared pointer to a variable of the desired memory class, *bytes* is the requested number of bytes, and *class\_name* is one of `THREAD_PRIVATE_MEM`, `NODE_PRIVATE_MEM`, `NEAR_SHARED_MEM`, `FAR_SHARED_MEM`, or `BLOCK_SHARED_MEM`; these symbolic constants are defined in `spp_prog_model.h`. Note that *memcls\_ptr* need not be of the class indicated in *class\_name*, allowing you to allocate one class of memory which is accessed by a pointer of a different class. This is analogous to using the `C$DIR class_name_POINTER` Fortran directive to allocate a pointer of one class to an array of a different class.

Not all combinations of pointer classes with data classes are supported, and not all make sense. Observe the following general rules:

- `thread_private` memory must be referenced by `thread_private` pointers.
- `node_private` memory should be referenced by `near_shared` or `far_shared` pointers.
- when shared data objects are referenced by private pointers, direct access to the object is restricted by the scope of the pointer. For example, if a `thread_private` pointer is used, access is restricted to the thread from which the pointer was allocated. If a `node_private` pointer is used, access is restricted to the threads on the hypernode from which the pointer was allocated. To access shared data using a `thread_private` pointer from a thread other than the thread that allocated the pointer, you must pass the pointer between threads. `node_private` pointers must similarly be passed between hypernodes when they are used.
- allocatable `block_shared` arrays are never explicitly assigned a pointer in Fortran.
- using shared-class pointers to point to shared-class memory provides pointer access to all threads, but, as with any

shared-memory data, appropriate latency rules apply to the pointers.

- in Fortran, when one of the `class_name_POINTER` directives is not used, dynamically allocated memory is accessed via a pointer of the same class as the allocated data.

Table 4 shows proper and improper pointer/data class combinations.

**Table 4** Pointer class/data class combinations

Pointer class	Data class				
	thread_private	node_private	near_shared	far_shared	block_shared
thread_private_pointer	OK	NR	OK	OK	OK
node_private_pointer	NR	OK	OK	OK	OK
near_shared_pointer	NR	OK	OK	OK	OK
far_shared_pointer	NR	OK	OK	OK	OK

In Table 4, “OK” means the pointer/data class combination is acceptable; “NR” means the combination is not recommended.

While pointer classes are provided for private variables, they are typically only needed when allocating shared memory.

### Default classes for dynamic memory

Using standard `malloc` in a shared memory C program will allocate `far_shared` memory. Memory allocated by either standard `malloc` or `memory_class_malloc` can be deallocated using the `free` function.

In Fortran, memory allocated using the `ALLOCATE` statement and not specifically assigned a class will be assigned the `far_shared` class by default. Such memory is deallocated using the `DEALLOCATE` statement. Refer to Appendix C, “Fortran 90 compatibility,” of the *Fortran Language Reference, 11th Edition*, for more information.

The stack can exist in only one physical space, so it is allocated in `near_shared` memory by default. This means that all local data (i.e. within a C function, all data not assigned a storage class of `static` or `extern`; in Fortran, all data not declared in `common`

blocks or `SAVED`) is `near_shared` and resides on hypernode 0 by default. This default can be changed to `far_shared` using the `mpa(1)` utility. For more information, refer to the `mpa(1)` man page.

Using each dynamic memory class is explained in detail in the sections that follow.

### `thread_private`

Since only the allocating thread can access dynamically allocated `thread_private` memory, it should be allocated (and deallocated) inside thread-parallel constructs, where each thread can allocate its own copy.

Consider the following Fortran example:

```
REAL*8 XTP(:)
C$DIR THREAD_PRIVATE(XTP)
      ALLOCATABLE(XTP)
      .
      .
      .
C$DIR LOOP_PARALLEL(THREADS), LOOP_PRIVATE(J)
      DO I = 1, NUM_THREADS()
C      THE FOLLOWING CODE MUST OCCUR INSIDE A
C      THREAD-PARALLEL CONSTRUCT
          ALLOCATE(XTP(N))
          DO J = 1, N
              . !COMPUTATIONS USING XTP
              .
              .
          ENDDO
          DEALLOCATE(XTP)
      ENDDO
```

This example assumes that the `ALLOCATE` statement is contained within a manually-created parallel loop or task. Then each parallel thread would get a private copy of the `XTP` array, of size `N` elements. Note the nested construct, in which the outer loop allocates and deallocates the `thread_private` array, and the inner loop uses it in computation. The outer loop calls the `NUM_THREADS()` intrinsic, which returns the number of threads on which the process is running, as described in Chapter 6, “Advanced shared memory programming.”

If this code was executed inside a node-parallel construct, each thread on each hypernode would allocate `XTP`.

If the `ALLOCATE` in this example was executed in a nonparallel section of code, only thread 0 would be able to reference `XTP`.

A similar C example follows:

```
static thread_private double *xtp;
.
.
.
#pragma _CNX loop_parallel(threads, ivar=i), loop_private(j)
for(i=0;i<num_threads();i++) {
/* the following statement must occur inside a
   parallel construct */
xtp=(double *)memory_class_malloc(sizeof(double)*n,
                                  THREAD_PRIVATE_MEM);

for(j=0;j<n;j++) {
. /*computations using xtp*/
.
.
}
free(xtp);
}
```

This example allocates memory in the same manner as the Fortran example.

#### **node\_private**

Recall that all threads access `node_private` memory via the same virtual addresses, and that these virtual addresses map to different physical addresses on each hypernode. Dynamic allocations of `node_private` memory should be executed in serial code. If you require physical memory on every hypernode and wish to reference it using the same virtual addresses from every hypernode, you can allocate the memory from serial code using a shared pointer. Allocation examples are given later in this section.

In order to access dynamically allocated `node_private` memory from `node-parallel` code, a shared pointer is needed. This is because when serial Fortran code running on hypernode 0 executes the `ALLOCATE` statement, the default `node_private` array pointer is assigned on that hypernode only. It is true that the virtual address maps to physical memory on every hypernode, but only the pointer of the assigning hypernode gets a value; the pointers on other hypernodes are unassigned. If hypernode 0 is the only hypernode requiring access to the array, all the threads running on it will be able to access the array via this `node_private` pointer. However, since the physical memory associated with the other hypernodes' pointers has not been assigned, attempting to use the values it contains will cause an error. Therefore, arrays dynamically allocated in serial code which are to be accessed in parallel code must be accessed with

explicitly-declared `near_shared` or `far_shared` pointers. Since the allocation takes place in serial code, there is no advantage to using one shared class over the other; both will be stored in physical memory on hypernode 0, causing pointer accesses from other hypernodes to take longer.

In Fortran, pointers default to the same memory class as the objects to which they point, so compiler directives are used to assign different memory classes to pointers. In C, the pointers must be explicitly declared separately, so the memory class assignment is handled in the pointer declaration.

Consider the following Fortran example:

```
REAL*8 XNP(:)
C$DIR NODE_PRIVATE(XNP)
C$DIR FAR_SHARED_POINTER(XNP)
ALLOCATABLE(XNP)
.
.
.
C THE FOLLOWING CODE SHOULD OCCUR OUTSIDE
C ANY PARALLEL CONSTRUCT:
ALLOCATE(XNP(1000))
```

This example assumes that the `ALLOCATE` statement is contained within a nonparallel section of code. Assuming this code is running on a 4-hypernode subcomplex, when the `ALLOCATE` statement executes, 8,000 bytes (1000 elements  $\times$  8 bytes per element) of virtual space are allocated for `XNP`. If the array is then accessed from a node-parallel construct, a unique physical copy of `XNP` will be created on any accessing hypernodes, and each accessing thread will access its hypernode's copy of `XNP`. If every hypernode on the subcomplex accesses the array, it will occupy a total of 32,000 bytes of physical memory.

A similar C example follows:

```
static far_shared double *xnp;
.
.
.
/* the following statement should occur outside any
   parallel construct: */
xnp = (double *)memory_class_malloc(sizeof(double)*1000,
                                   NODE_PRIVATE_MEM);
```

This example allocates memory in the same manner as the Fortran example. Note that the pointer to `xnp` is explicitly assigned the `far_shared` class in its declaration.

The preceding examples allow you to use the same names to access physically unique `node_private` arrays on each hypernode from node-parallel code. This can effectively increase your program's virtual memory space, since the same amount of virtual space is used for the arrays no matter how many hypernodes hold physical copies or run code that accesses them.

Parallel allocations of `node_private` memory should normally be done outside parallel code. This memory should then be accessed via shared pointers. When memory is allocated in such a fashion, threads will only be able to access the data which is stored physically on their hypernode. To access `node_private` data that is stored on another hypernode, the data must be passed via a shared variable.

These allocations are useful when private work arrays are needed by each hypernode in a node-parallel construct.

Consider the following Fortran example:

```

REAL*8 NODE_V(:)
ALLOCATABLE(NODE_V)
C$DIR NODE_PRIVATE(NODE_V)
C$DIR FAR_SHARED_POINTER(NODE_V)
.
.
.
NN = NUM_NODES()
JLMT = (JTOT/NN) + 1
ALLOCATE(NODE_V(JLMT))
C$DIR LOOP_PARALLEL(NODES, CHUNK_SIZE = 1)
C$DIR LOOP_PRIVATE(J)
DO I = 1, M
C$DIR LOOP_PARALLEL(THREADS)
DO J = 1, JLMT
    NODE_V(J) = PI * (RAD(J)**2) * L(I, J)
    P(I, J) = (N(I, J) * R * T(I, J)) / NODE_V(J)
ENDDO
C$DIR LOOP_PARALLEL(THREADS)
DO J = 2, JLMT
    IF ((NODE_V(J) .GT. NODE_V(J-1)) .AND.
^      (NODE_V(J) .GT. NODE_V(J+1))
^      LMAX_V(I, J) = NODE_V(J)
ENDDO
.
.
.
ENDDO
DEALLOCATE(NODE_V)

```

Here, the `NODE_V` array is used privately by each hypernode in the computation of the array `P` and to find values for `LMAX_V`. `NODE_V` is not needed outside of the `I` loop. Note that the `NUM_NODES()` intrinsic (described in Chapter 6, “Advanced shared memory programming”) is used to determine the number of hypernodes on which the process is running.

An analogous C example follows:

```
static far_shared double *node_v;
.
.
.
nn = num_nodes();
jlmnt = (jtot/nn) + 1;
node_v = (double*)memory_class_malloc(jlmnt*sizeof(double),
                                     NODE_PRIVATE_MEM);
#pragma _CNX loop_parallel(nodes, chunk_size = 1, ivar = i)
#pragma _CNX loop_private(j)
for(i=0; i<m; i++) {
    #pragma _CNX loop_parallel(threads)
    for(j=0; j<jlmnt; j++) {
        node_v[j] = pi*rad[j]*rad[j]*l[i,j];
        p[i,j] = (n[i,j]*r*t[i,j])/node_v[j];
    }
    #pragma _CNX loop_parallel(threads)
    for(j=1; j<jlmnt; j++) {
        if((node_v[j] > node_v[j-1]) && (node_v[j] > node_v[j+1]))
            lmax_v[i,j] = node_v[j];
    }
}
free(node_v);
```

### **near\_shared**

To be most useful, `near_shared` memory must be dynamically allocated on the hypernode that will most heavily access it. `near_shared` memory should be allocated from within explicitly-defined node-parallel structures to ensure that it is allocated on the hypernodes that will use it most. Node-parallel structures are manually defined by the programmer using the `loop_parallel(nodes)` or `begin_tasks(nodes)` directive and `pragma`, which are discussed in detail in Chapter 4, “Basic shared memory programming.”

Unconditionally allocating `near_shared` arrays in node-parallel structures is possible and is discussed later. However, to take full advantage of `near_shared` arrays, they should be allocated conditionally, based on the allocating hypernode, within node-parallel code.

A Fortran example of this follows. For simplicity, we assume this example will always run on a 2-hypernode subcomplex.

```
      REAL*8 XNS(:), YNS(:)
      ALLOCATABLE(XNS, YNS)
C$DIR NEAR_SHARED(XNS, YNS)
      .
      .
      .
C THE FOLLOWING CODE MUST RUN NODE-PARALLEL
C$DIR LOOP_PARALLEL(NODES) !2-NODE SUBCPLX
C$DIR LOOP_PRIVATE(NODE_ID)
      DO I = 1, NUM_NODES()! IS ASSUMED
          NODE_ID = MY_NODE()
          IF(NODE_ID .EQ. 0) THEN
              ALLOCATE(XNS(1000))
          ELSE
              ALLOCATE(YNS(1000))
          ENDIF
      .
      .
      .
      ENDDO
```

The `LOOP_PARALLEL(NODES)` directive is used to achieve node-parallelism; the `MY_NODE()` function returns the hypernode ID of the hypernode on which the current thread is executing. These are both discussed in more detail in Chapter 6, “Advanced shared memory programming.” This example allocates a near\_shared `XNS(1000)` array for hypernode 0, and a near\_shared `YNS(1000)` array for hypernode 1. When accessed, `XNS` is stored in hypernode-global physical memory on hypernode 0, and is therefore accessible from that hypernode with the least latency; similarly, `YNS` is physically stored on and quickly accessible from hypernode 1. The arrays’ unique virtual addresses allow either to be accessed by the hypernode it doesn’t occupy, and more latency is involved in such accesses. This example presumes that code that follows in the program conditionally manipulates these arrays on their respective hypernodes, but that some sharing of the array data between hypernodes also occurs.

A C example which allocates memory in the same fashion follows. Again we assume that this code will always run on a two-hypernode subcomplex.

```
static near_shared double *xns, *yns;
int node_id;
.
.
.
/* the following code must run node-parallel on a 2-node
   subcomplex: */
#pragma _CNX loop_parallel(nodes, ivar = i), loop_private(node_id)
for(i=0;i<num_nodes();i++) {
  node_id = my_node();
  if(node_id == 0)
    xns = (double *)memory_class_malloc(sizeof(double)*1000,
                                         NEAR_SHARED_MEM);
  else
    yns = (double *)memory_class_malloc(sizeof(double)*1000,
                                         NEAR_SHARED_MEM);
.
.
.
}
```

Note that in C, the pointer class is assigned with the type declaration, and the data class is assigned by the `memory_class_malloc` function.

Recall that `near_shared` data objects have a single virtual address used by all hypernodes, and, when accessed, a single physical address (on the allocating hypernode). In the above examples, the pointers used to access the `near_shared` arrays were of the same memory class as the arrays (by default in Fortran and by explicit typing in C). The allocations are not thread-parallel, so a single thread on each hypernode will allocate its respective array. If at some point later in the program the code goes thread-parallel, all threads on a given hypernode will be able to access the arrays via their `near_shared` pointers.

However, if you wish to allocate `near_shared` memory from within a thread-parallel structure, the `near_shared` pointer can present a problem when the `near_shared` data space is actually allocated: all threads will be allocating the space using the same shared pointer, so each thread's `ALLOCATE` will reset the pointer. In C the pointer class is directly controllable, but in Fortran the `memory_class_POINTER` directive must be used to assign a pointer of a different class to the array.

Table 4 on page 157 covers acceptable pointer classes to use for `near_shared` data. While a certain combination may be allowed, it may not make sense for the task at hand. In the following Fortran example, the pointer to ZNS is assigned the `THREAD_PRIVATE` memory class, because ZNS is being allocated in a thread-parallel structure. This causes each thread to allocate its own `near_shared` copy of ZNS on its hypernode.

```

      REAL*8 ZNS ( : )
      ALLOCATABLE ( ZNS )
C$DIR NEAR_SHARED ( ZNS )
C$DIR THREAD_PRIVATE_POINTER ( ZNS )
      .
      .
      .
C THE FOLLOWING CODE MUST RUN THREAD-PARALLEL
C$DIR LOOP_PARALLEL ( THREADS )
      DO I = 1, NUM_THREADS
          ALLOCATE ( ZNS ( 1000 ) )
          .
          .
          .
      ENDDO

```

Here, thread-parallelism is achieved using the `LOOP_PARALLEL ( THREADS )` directive, which is discussed further in Chapter 4, “Basic shared memory programming.” The hypernode on which the `ALLOCATE` statement executes allocates a `near_shared` copy of ZNS for each thread. Each thread running on that hypernode can then reference its copy of ZNS via the `thread_private` pointers provided by the `THREAD_PRIVATE_POINTER` directive on ZNS. If the thread-parallel section of code shown is also running node-parallel, each hypernode running it will allocate as many `near_shared` ZNS arrays as it has threads. For example, if 8 threads are running on each of 2 hypernodes, 16 physical and virtual copies of ZNS will be created. However, because of the thread-private pointers, these `near_shared` copies will lose their direct-accessibility; accesses by any thread other than the allocating thread is only possible in C, and will require sharing of the `thread_private` pointers.

A similar C example follows:

```
static thread_private double *zns;
.
.
.
/* the following code must run thread-parallel */
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0;i<num_threads();i++)
    zns = (double *)memory_class_malloc(sizeof(double)*1000,
                                         NEAR_SHARED_MEM);
.
.
.
```

This example allocates memory in the same manner as the Fortran example.

When only one parallel hypernode in a group of hypernode-parallel tasks needs a private array, the `near_shared` class can be allocated dynamically from within the task in question, a `near_shared` pointer can be used to provide low-latency access to the array.

The following Fortran code shows a parallel allocation for a single parallel hypernode:

```
REAL*8 NODE_SCRATCH(:)
ALLOCATABLE(NODE_SCRATCH)
C$DIR NEAR_SHARED(NODE_SCRATCH)
.
.
.
C$DIR BEGIN_TASKS(NODES), TASK_PRIVATE(I)
DO I = 1, N
    A(I) = B(I) + C(I)
ENDDO
C$DIR NEXT_TASK
CALL TSUB(X,Y)
.
.
.
```

```

C$DIR NEXT_TASK
    ALLOCATE (NODE_SCRATCH(1000))
C$DIR LOOP_PARALLEL (THREADS)
    DO I = 1, 1000
        NODE_SCRATCH(I) = Z(I)
    ENDDO
C$DIR LOOP_PARALLEL (THREADS)
    DO I = 1, 999
        Z(I) = NODE_SCRATCH(I+1)
        .
        .
        .
    ENDDO
    DEALLOCATE (NODE_SCRATCH)
C$DIR END_TASKS
    .
    .
    .

```

Here, the final task in the list allocates the temporary near\_shared work array `NODE_SCRATCH`, uses it, and deallocates it. The compiler will thread-parallelize both loops within this task because of the `LOOP_PARALLEL` directives on them, and they will both be able to access the `NODE_SCRATCH` array with minimal latency. Since this array is near\_shared, it can also be accessed by any other hypernode, though that doesn't happen in this example. Using `LOOP_PARALLEL` to manually parallelize loops is further discussed in Chapter 6, "Advanced shared memory programming."

An analogous C example follows:

```

static near_shared double *node_scratch;
.
.
.
#pragma _CNX begin_tasks(nodes), task_private(i)
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
}
#pragma _CNX next_task
tsub(x,y);
.
.
.

```

```

#pragma _CNX next_task
node_scratch = (double *)memory_class_malloc(1000*sizeof(double),
                                             NEAR_SHARED_MEM);
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0; i<1000; i++) {
    node_scratch[i] = z[i];
}
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0; i<999; i++) {
    z[i] = node_scratch[i+1];
    .
    .
    .
}
free(node_scratch);
#pragma _CNX end_tasks
.
.
.

```

### **far\_shared**

Since `far_shared` memory is distributed round-robin across all hypernodes in the subcomplex, it is best dynamically allocated in nonparallel structures. Allocating `far_shared` memory in a thread- or node-parallel structure would create multiple copies of the requested `far_shared` array, one for each allocating thread or hypernode in the structure. This replication increases the amount of memory, both physical and virtual, used by the process, and is of questionable utility.

Consider the following Fortran example:

```

REAL*8 XFS(:)
ALLOCATABLE(XFS)
C$DIR FAR_SHARED(XFS)
.
.
.
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
ALLOCATE(XFS(10000))

```

Since the `far_shared` data in this example is allocated outside of parallel code, the default `far_shared` pointer is suitable; the array is allocated once, and there is no danger of parallel allocations resetting the pointer. When the `ALLOCATE` statement executes, 80,000 bytes of virtual space is allocated. When the array is accessed, this maps to 80,000 bytes of physical memory, which

is distributed round-robin by 4KB pages to each hypernode in the subcomplex.

A similar C example follows:

```
static far_shared double *xfs;
.
.
.
/* the following code should run in a nonparallel
   structure: */
xfs = (double *)memory_class_malloc(sizeof(double)*10000,
                                     FAR_SHARED_MEM);
```

This example allocates memory in the same manner as the Fortran example.

Keep in mind that even dynamically allocated `far_shared` memory is distributed across all hypernodes in the subcomplex, not just across the hypernodes on which your program can spawn threads. Therefore, if your program is constrained to a subset of the hypernodes available in its subcomplex, you should use the `block_shared` or `near_shared` memory classes to avoid placing data on unused hypernodes.

`far_shared` is the default memory class for all data not otherwise assigned a class.

#### **block\_shared**

`block_shared` memory is specifically provided for dynamic allocation by a process running on multiple hypernodes. It is ideal for array-manipulating loops which will parallelize across hypernodes, with each hypernode computing chunks of contiguous elements of the arrays. Using `block_shared` memory for such arrays distributes the chunks across the hypernodes on which the program is running; work can then be distributed so that each hypernode accesses the elements that reside on it most frequently. If necessary, hypernodes can still directly access each other's `block_shared` data.

As with `far_shared` data, this `block_shared` data allocation should take place outside of parallel structures; in this case, the compiler will automatically distribute the data evenly across the hypernodes of the subcomplex. Allocating `block_shared` memory from parallel structures will cause multiple copies (one per parallel thread) of the `block_shared` data to be created, a situation that wastes memory and is of questionable utility.

Consider the following Fortran example:

```
REAL*8 XBS(:), YBS(:)
ALLOCATABLE(XBS, YBS)
C$DIR BLOCK_SHARED(XBS, YBS)
.
.
.
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
    ALLOCATE(XBS(10240), YBS(7680))
```

Here, 81920 bytes of virtual space is requested for XBS, and 61440 bytes is requested for YBS. Assuming this code is running on a 4-hypernode subcomplex, these `block_shared` arrays will have their physical pages divided equally, in contiguous chunks, among the 4 hypernodes. Recall that the page size is 4096 bytes, so XBS occupies 20 pages ( $81920/4096 = 20$ ). The number of hypernodes, 4, is an integral divisor of 20, giving 5 pages per hypernode. So the first 5 pages of XBS are physically mapped to hypernode 0 (yet still accessible via their virtual addresses from any other hypernode), the second 5 pages go to hypernode 1, the third 5 pages to hypernode 2, and the last 5 pages to hypernode 3.

The 61440 bytes of YBS, on the other hand, occupy 15 pages. 4 does not integrally divide 15, so the compiler automatically increases the size of YBS just enough to allow the number of hypernodes to integrally divide its pages. YBS becomes an 8192 element array; it now occupies 65536 bytes or 16 pages of memory. These 16 pages are divided up so that the first 4 pages map to hypernode 0 and so on, with the last 4 mapping to hypernode 3.

Any hypernode-parallel code that follows the above allocation should be written such that the portion running on a particular hypernode accesses the array elements resident on that hypernode. For example, the code running on hypernode 2 should make most frequent use of XBS(5121:7680) and YBS(3841:5760). The "Accessing hypernode-local `block_shared` elements" section, which follows, discusses how to determine which elements of a `block_shared` array are on a given hypernode.

A similar C example follows:

```
static near_shared double *xbs, *ybs;
.
.
.
/* the following code should be run in a nonparallel
   section of code: */
xbs = (double *)memory_class_malloc(sizeof(double)*10240,
                                   BLOCK_SHARED_MEM);
ybs = (double *)memory_class_malloc(sizeof(double)*7680,
                                   BLOCK_SHARED_MEM);
```

This example allocates and distributes `xbs` and `ybs` as `block_shared` arrays exactly as the Fortran example did, including the automatic resizing of `ybs.near_shared` pointers are used to access the arrays.

When allocated, `block_shared` arrays are distributed across all hypernodes available to your program; you cannot constrain the number of hypernodes that the arrays occupy. This makes the `block_shared` class especially suitable for programs whose data sets scale with the number of available hypernodes. If there is any question as to whether your `block_shared` arrays will occupy enough pages to efficiently use the number of hypernodes available to your program, you should roughly compute the number of pages the array in question is likely to occupy. If this number is not at least equal to the number of hypernodes that will typically be available to the program, you can still use the `block_shared` class, but your program might waste some memory by expanding the array to occupy at least one page per hypernode.

While you cannot constrain the allocation of `block_shared` memory from within your program, you can constrain the number of hypernodes on which your program runs by using the `mpa(1)` utility or the `max_threads loop_parallel` attribute. `block_shared` memory is only allocated on the hypernodes on which your program can spawn threads, so if your program is constrained in this way, `block_shared` memory is preferable to `far_shared` memory, which is distributed across all hypernodes in the subcomplex.

Because `block_shared` data elements each have a unique virtual and physical address (there is no virtual address replication as with the `node_private` class), their size is limited by the 4 Gbyte virtual address space limit. Keep this in mind if your data set size scales with the number of available hypernodes. If your program needs to surpass this limit, it can do so using the `node_private` class as described in the "Advanced shared

memory example" of Chapter 6, "Advanced shared memory programming."

### Accessing hypernode-local `block_shared` elements

Determining the range of `block_shared` array element indexes located on a given hypernode is easily computable given the page size, number of hypernodes, element size, and number of elements. The following Fortran function takes these parameters along with the current hypernode number as arguments and returns the base index for the elements on the current hypernode.

```
INTEGER FUNCTION MIN_NODE_ELT (PGSZ, NN, ELTSZ, NELTS, CUR_NODE)
INTEGER PGSZ, NN, ELTSZ, NELTS, CUR_NODE
INTEGER NUM_PGS, PGS_PER_NODE
NUM_PGS = 1 + (NELTS*ELTSZ - 1)/PGSZ
PGS_PER_NODE = 1 + (NUM_PGS - 1)/NN
C ADJUST MIN_NODE_ELT BY +1 TO COMPENSATE FOR ARRAY INDEXING
C FROM 1:
MIN_NODE_ELT = (CUR_NODE*PGS_PER_NODE*PGSZ/ELTSZ) + 1
RETURN
END
```

This Fortran example assumes the array is indexed from 1.

The analogous C function, which assumes indexing begins at 0, is shown below.

```
int min_node_elt(int pgsz,int nn,int eltsz,int nelts,int cur_node)
{
    int num_pgs, pgs_per_node;
    num_pgs = 1 + (nelts*eltsz - 1)/pgsz;
    pgs_per_node = 1 + (num_pgs - 1)/nn;
    return (cur_node*pgs_per_node*pgsz/eltsz);
}
```

The following Fortran example shows one way in which `MIN_NODE_ELT` can be used in a hypernode-parallel loop so that each hypernode accesses only its local array elements. This example assumes that the number of pages occupied by XBS is large enough to efficiently exploit all available hypernodes, and that the 4 Gbyte virtual address limit is not surpassed.

```

REAL*8 XBS(:)
ALLOCATABLE(XBS)
C$DIR BLOCK_SHARED(XBS)
.
.
.
C **NO PARALLELISM**
ALLOCATE(XBS(AR SZ))
NN = NUM_NODES()
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J, MN, MIN_ELT, MAX_ELT)
DO I = 1, NN ! DO ON EACH NODE:
    MN = MY_NODE() ! GET CURRENT NODE NUMBER
C    GET MIN AND MAX ELEMENT NUMBERS FOR CURRENT NODE:
    MIN_ELT = MIN_NODE_ELT(PGSZ, NN, 8, AR SZ, MN)
    MAX_ELT = MIN_NODE_ELT(PGSZ, NN, 8, AR SZ, MN+1) - 1
C    GO THREAD PARALLEL ON CURRENT NODE:
C$DIR LOOP_PARALLEL(THREADS)
    DO J = MIN_ELT, MAX_ELT ! LOOP OVER LOCAL ELEMENTS
        XBS(J) = ...
    .
    .
    .
ENDDO
ENDDO

```

Here, the array XBS is allocated in serial code. When the program goes hypernode-parallel, each iteration calls MY\_NODE to get its hypernode ID (which ranges from 0..NUM\_NODES), then uses this ID in the following calls to MIN\_NODE\_ELT to determine the minimum and maximum indexes of the hypernode-local elements of XBS. Each hypernode then computes its elements of XBS in a thread-parallel loop that iterates over only the resident elements of XBS.

The analogous C code follows:

```

static near_shared double *xbs;
.
.
.
/***** no parallelism *****/
xbs = (double *)memory_class_malloc(sizeof(double)*arsz,
                                   BLOCK_SHARED_MEM);
nn = num_nodes();

```

```

#pragma _CNX loop_parallel(nodes, ivar = i)
#pragma _CNX loop_private(j, mn, min_elt, max_elt)
for(i=0; i<nn; i++) { /* do on each node: */
  mn = my_node(); /* get current node number */
  /* get min and max element numbers for current node: */
  min_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn);
  max_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn+1) - 1;
  /* go thread parallel on current node: */
#pragma _CNX loop_parallel(threads, ivar = j)
  for(j=min_elt; j<=max_elt; j++) { /* loop over local elts */
    xbs[j] = ...
    .
    .
    .
  }
}
}

```

Another easy way to access hypernode-local elements is to add a dimension of size `1..num_nodes()` to your `block_shared` array when you allocate it.

The following Fortran example shows such an allocation:

```

REAL*8 ABS(,:)
ALLOCATABLE(ABS)
C$DIR BLOCK_SHARED(ABS)
.
.
.
C **NO PARALLELISM**
  ALLOCATE(ABS(N,NUM_NODES()))
.
.
.
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
DO I = 1, NUM_NODES()
C$DIR LOOP_PARALLEL(THREADS)
  DO J = 1, N
    ABS(J,I) = ...
    .
    .
    .
  ENDDO
ENDDO

```

Here, `N` is chosen so that `N*NUM_NODES()` is equal to or greater than the total number of `ABS` elements required. We assume that the original problem (before it is rewritten for parallelization) does not require a two-dimensional array, and that the second dimension is provided only to allow each parallel hypernode to

easily index its local elements. Inside the  $\mathcal{J}$  loop, the  $\mathcal{I}$  index into ABS ensures that each parallel hypernode accesses its local elements automatically.



---

# Advanced shared memory programming

# 6

Most of the manual parallelization techniques discussed in Chapter 4, “Basic shared memory programming,” allow you to take advantage of the compilers’ automatic dependence checking and data privatization. The examples which used the `LOOP_PRIVATE` and `TASK_PRIVATE` directives and pragmas are exceptions to this; in these cases, manual privatization was required, but it was done on a loop-by-loop basis. Only the simplest data dependencies were handled in Chapter 4.

This chapter is concerned with manual parallelizations that use the program-wide memory classes discussed in Chapter 5, “Memory classes,” and that handle multiple and ordered data dependencies.

Before we can discuss specific examples of such parallelization, however, we must introduce the remaining underlying concepts and available functions.

---

## Parallel information functions

Several SPP Series intrinsics are available to provide information regarding the parallelism or potential parallelism of your program. These are all integer functions, available in both 4- and 8-byte lengths, and can appear in executable statements anywhere an integer expression is legal. The 8-byte versions, which are suffixed with `_8`, are typically only used in Fortran programs in which the default data lengths have been changed using the `-cfc`, `-p8` or similar compiler options. These versions expect 8-byte input arguments and return 8-byte values.

## Note

All C code examples presented in this chapter assume that the line `#include <spp_prog_model.h>` appears above the C code presented. This header file contains the necessary type and function definitions.

The subsections that follow describe these functions.

---

## Number of processors

These functions return the total number of processors on which the process has initiated threads. These threads are not necessarily active.

In Fortran, these functions have the following forms:

```
INTEGER NUM_PROCS()  
INTEGER*8 NUM_PROCS_8()
```

In C, they have the following forms:

```
int num_procs(void);  
long long num_procs_8(void);
```

`num_procs` can be used to dimension automatic and adjustable arrays in Fortran, and may be used in C or Fortran to dynamically specify array dimensions and allocate storage.

---

## Number of threads

These functions return the total number of threads the process creates at initiation, regardless of how many hypernodes the threads occupy, and regardless of how many are idle or active. They are typically used to manually define thread-parallel loops which may span hypernodes.

In Fortran, these functions have the following forms:

```
INTEGER NUM_THREADS()  
INTEGER*8 NUM_THREADS_8()
```

In C, they have the following forms:

```
int num_threads(void);  
long long num_threads_8(void);
```

The return value will only differ from `num_procs` if threads are oversubscribed.

---

## Number of hypernodes

These functions return the number of hypernodes on which the process is running. They can be used to dimension automatic and adjustable arrays in Fortran, and can be used in both C and Fortran to dynamically specify array dimensions and allocate storage.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODES ()
INTEGER*8 NUM_NODES_8 ()
```

In C, they have the following forms:

```
int num_nodes(void);
long long num_nodes_8(void);
```

## Number of threads on current hypernode

These functions return the number of the calling process's threads running on the hypernode from which the function is called. This number can vary from one hypernode to another depending on subcomplex configurations, usage of manual parallelization directives, and the number of processors installed on each hypernode.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODE_THREADS ()
INTEGER*8 NUM_NODE_THREADS_8 ()
```

In C, they have the following forms:

```
int num_node_threads(void);
long long num_node_threads_8(void);
```

## Thread ID

When called from parallel code these functions return the spawn thread ID of the calling thread, in the range 0..*nst*-1, where *nst* is the number of threads in the current spawn context (the number of threads spawned by the last parallel construct). Use them when you wish to direct specific tasks to specific threads inside parallel constructs.

In Fortran, these functions have the following forms:

```
INTEGER MY_THREAD ()
INTEGER*8 MY_THREAD_8 ()
```

In C, they have the following forms:

```
int my_thread(void);
long long my_thread_8(void);
```

When called from code which is not running parallel due to compiler parallelism, `cps_ppcall()`, or `cps_ppcalln()` (e.g. serial code or code that is parallel due to asymmetric CPSlib calls), these functions return 0.

---

## Hypernode ID

These functions return the logical hypernode ID of the hypernode on which the calling thread is running, in the range `0..num_nodes()-1`. Use them when you wish to direct specific tasks to specific hypernodes inside parallel constructs.

In Fortran, these functions have the following forms:

```
INTEGER MY_NODE()  
INTEGER*8 MY_NODE_8()
```

In C, they have the following forms:

```
int my_node(void);  
long long my_node_8(void);
```

Logical hypernode IDs range from  $0..n-1$ , where  $n$  is the number of available hypernodes in the subcomplex. Logical IDs are assigned in the order in which your program occupies the subcomplex. The hypernode that your program's thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because SPP-UX starts a program on the least-loaded hypernode, mapping of logical hypernode IDs to physical hypernodes can differ between programs due to load balancing; thus two programs running on the same subcomplex are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs have no correlation to physical hypernode IDs, which are unique for each hypernode at the machine level.

---

## Level of parallelism

These functions return a value representing the level of parallelism of the calling process.

In Fortran, these functions have the following forms:

```
INTEGER LEVEL_OF_PARALLELISM()  
INTEGER*8 LEVEL_OF_PARALLELISM_8()
```

In C, they have the following forms:

```
int level_of_parallelism(void);  
long long level_of_parallelism_8(void);
```

The return value is one or a sum of the values shown in Table 5. In C, these values are #defined as symbolic constants in `spp_prog_model.h`.

Table 5 Levels of parallelism

Function return value	C symbolic constant name	Meaning
0	CPS_PL_NONE	Not parallel
1	CPS_PL_PARALLEL	Asymmetric thread active
2	CPS_PL_NODE	Node-parallelism
4	CPS_PL_NTHREAD	Thread-parallelism within a hypernode
8	CPS_PL_THREAD	Single-dimensional thread-parallelism

As an example of how these can be summed, assume the return value is 6. This means the process is two-dimensionally parallel; it first went parallel across hypernodes, and within the current hypernode it went parallel again on the threads of the hypernode. This differs from a return value of 8, which means the process went one-dimensionally thread-parallel, and occupies all available threads on all available hypernodes with no nested parallelism.

The valid sum values are: 3,5,6,7, and 9.

A return value of 1, or a sum including 1, means an asymmetric thread is active in the calling program. Asymmetric parallelism is currently only supported by the Compiler Parallel Support Library. Refer to Appendix D for more information.

---

## Stack memory type

These functions return a value representing the memory class that the current thread stack is allocated from. The thread stack holds all the procedure-local arrays and variables not manually assigned a class. The thread stack is created in `near_shared` memory by default, but this can be changed via the `mpa(1)` utility.

In Fortran, these functions have the following forms:

```
INTEGER MEMORY_TYPE_OF_STACK()
INTEGER*8 MEMORY_TYPE_OF_STACK_8()
```

In C, they have the following forms:

```
int memory_type_of_stack(void);
long long memory_type_of_stack_8(void);
```

These functions return one of the values described in Table 6.

**Table 6** Stack type return values

Function return value	C symbolic constant name	Stack memory type
4	FAR_SHARED_MEM	far_shared
3	NEAR_SHARED_MEM	near_shared
2	NODE_PRIVATE_MEM	node_private

---

## Thread IDs and nested parallelism

As discussed in Chapter 4, “Basic shared memory programming,” you can manually parallelize nested loops and tasks to exploit up to two dimensions of parallelism. If you choose to do this, the first dimension must be node-parallel and the second must be thread-parallel. If thread-parallelism is exploited first, no dimensions are left; it is a programming error to attempt to spawn node-parallelism from within a thread-parallel construct. However, single-dimensional thread-parallel code can exploit all the threads on a subcomplex, even if they span hypernodes.

If you attempt to spawn thread-parallelism from within a thread-parallel construct, the compiler will ignore your directives, and your inner parallel construct will simply run serially.

---

## Thread ID assignments

Chapter 3, “Compiler optimizations,” discusses how programs are initiated as a collection of threads, one per available processor, and how all but thread 0 are idle until parallelism is encountered. We will now discuss the details of how threads are spawned and assigned IDs.

When a process begins, the threads created to run it have unique *kernel* thread IDs. Thread 0, which runs all the serial code in the program, has kernel thread ID 0; the rest of the threads have unique but unspecified kernel thread IDs at this point. The `num_threads()` intrinsic will return the number of threads created, regardless of how many are active when it is called.

When thread 0 encounters parallelism, it *spawns* some or all of the threads created at program start. This means it causes these threads to go from idle to active, at which point they begin working on their share of the parallel code. All available threads are spawned by default, but this can be changed using various compiler directives.

If the parallel structure is thread-parallel, then `num_threads()` threads will be spawned, subject to user-specified limits. At this point, kernel thread 0 becomes *spawn* thread 0, and the spawned threads are assigned spawn thread IDs ranging from `0..num_threads()-1` (this range begins at what used to be kernel thread 0). If you manually limit the number of spawned threads, these IDs will range from 0 to one less than your limit. If you attempt to spawn thread-parallelism within an already thread-parallel structure, the thread attempting to spawn will acquire spawn thread ID 0. If all threads attempt to spawn thread parallelism in this manner, they will all become spawn thread 0, each in a unique context.

If the parallel structure is node-parallel, then `num_nodes()` threads will be spawned, one per available hypernode, subject to user-specified limits. Again, kernel thread 0 becomes spawn thread 0, and in this case, the spawn thread IDs range from `0..num_nodes()-1`, subject to user limits as described above.

If thread-parallelism is then encountered within this node-parallelism, `num_node_threads()` threads will be spawned on the hypernode or hypernodes encountering the thread-parallelism. These spawned threads will have spawn thread IDs, which are specific to the hypernode they are running on, ranging from `0..num_node_threads()-1`, with spawn thread ID 0 belonging to the initial thread which executes the spawn. `num_node_threads()` may return a different value on each hypernode when called from node-parallel code.

Note that, with nested parallelism, a node-parallel thread that encounters a thread-parallel construct becomes spawn thread 0 on that hypernode regardless of its previous spawn thread ID. When this thread exits the thread-parallel construct, it returns to its previous spawn thread ID. The `my_thread()` intrinsic function returns the caller's spawn thread ID, which depends on the level of parallelism.

---

## Synchronization tools

The compiler cannot automatically parallelize loops containing dependencies, but a rich set of directives, pragmas and data types is available to help you manually parallelize such loops by synchronizing (and, if necessary, ordering) access to the code containing the dependency. These directives can also be used to synchronize dependencies in parallel tasks. They allow you to efficiently exploit parallelism in structures that would otherwise be unparallelizable.

---

### Gates and barriers

Gates allow you to restrict execution of a block of code to a single thread. They can be allocated, locked, unlocked and deallocated via the functions described in the “Synchronization functions” section, or they can be used with the ordered or critical section directives, which automate the locking and unlocking functions.

Barriers block further execution until all executing threads reach the barrier.

Gates and barriers use dynamically allocatable variables, declared using compiler directives in Fortran and using data type statements in C. They may be initialized and referenced only by passing them as arguments to the functions discussed in the following “Synchronization functions” section.

In C, gates and barriers are declared using the `gate_t`, `gate8_t`, `barrier_t` and `barrier8_t` data type statements, which have the following forms:

```
gate_t  namelist
gate8_t namelist
barrier_t namelist
barrier8_t namelist
```

where *namelist* is a comma-delimited list of one or more gate or barrier names, as appropriate. `gate8_t` and `barrier8_t` are used to declare 8-byte gate and barrier variables. The other declarations declare default-size variables.

In C, gates and barriers should appear only in definition and declaration statements, and as formal and actual arguments.

In Fortran, gates and barriers are declared using the `GATE` and `BARRIER` compiler directives, which have the following forms:

```
C$DIR GATE (namelist)
C$DIR BARRIER (namelist)
```

where *namelist* is a comma-delimited list of one or more gate or barrier names, as appropriate. These declare variables of the

appropriate size; separate 4- and 8-byte versions are not needed in Fortran. No other type declarations are necessary for these variables; the compiler directives alone are sufficient.

In Fortran, gates and barriers can only appear in `COMMON` and `DIMENSION` statements, in preceding type statements, and as dummy and actual arguments. Gate and barrier types override other types declared using the same names prior to the gate/barrier declaration. Once a variable is declared as a gate or barrier, it cannot be redeclared as another type. Gates and barriers cannot be equivalenced. If you place gates or barriers in common, the common block declaration must precede the `GATE` directive, and the common block must contain only gates or only barriers; it must not contain other data types. Arrays of gates or barriers must be dimensioned using `DIMENSION` statements.

---

## Synchronization functions

The C and Fortran allocation, deallocation, lock and unlock functions provided for use with gates and barriers are listed here. 4- and 8-byte versions are provided; the 8-byte Fortran functions are primarily for use with compiler options which change the default data size to 8 bytes (e.g. `-cfc`, `-p8`, `-pd8`). You must be consistent in your choice of versions—memory allocated using an 8-byte function must be deallocated using an 8-byte function.

Examples of using these functions are presented and explained in the “Synchronizing code” section, which follows.

### Allocation functions

These functions allocate memory for a gate or barrier. When first allocated, gate variables are unlocked.

The Fortran gate and barrier allocation functions have the following declarations:

```
INTEGER FUNCTION ALLOC_GATE (gate)
INTEGER*8 FUNCTION ALLOC_GATE_8 (gate)
INTEGER FUNCTION ALLOC_BARRIER (barrier)
INTEGER*8 FUNCTION ALLOC_BARRIER_8 (barrier)
```

Where *gate* and *barrier* are the gate or barrier variables, as appropriate. These variables must be declared as described in the “Gates and barriers” section of this chapter.

In C, the functions have the following declarations:

```
int alloc_gate(gate_t *gate_p);
long long alloc_gate_8(gate8_t *gate_p);
int alloc_barrier(barrier_t *barrier_p);
long long alloc_barrier_8(barrier8_t *barrier_p);
```

Where *gate\_p* and *barrier\_p* are pointers of the indicated type, which have been previously declared as described in the “Gates and barriers” section of this chapter.

### Deallocation functions

These functions free the memory assigned to the specified gate or barrier variable.

The Fortran gate and barrier deallocation functions have the following declarations:

```
INTEGER FUNCTION FREE_GATE(gate)
INTEGER*8 FUNCTION FREE_GATE_8(gate)
INTEGER FUNCTION FREE_BARRIER(barrier)
INTEGER*8 FUNCTION FREE_BARRIER_8(barrier)
```

Where *gate* and *barrier* are the previously-declared gate or barrier variables, as appropriate.

In C, the functions have the following declarations:

```
int free_gate(gate_t *gate_p);
long long free_gate_8(gate8_t *gate_p);
int free_barrier(barrier_t *barrier_p);
long long free_barrier_8(barrier8_t *barrier_p);
```

Where *gate\_p* and *barrier\_p* are previously-declared pointers of the indicated type.

Always free gates and barriers when you are done using them.

### Locking functions

These functions acquire a gate for exclusive access. If the gate cannot be immediately acquired, the calling thread waits for it. The conditional locking functions, which are prefixed with `COND_` or `cond_`, acquire a gate if doing so doesn’t require a wait. If the gate is acquired, the functions return 0; if not, they return -1.

The Fortran gate locking functions have the following declarations:

```
INTEGER FUNCTION LOCK_GATE(gate)
INTEGER*8 FUNCTION LOCK_GATE_8(gate)
INTEGER FUNCTION COND_LOCK_GATE(gate)
INTEGER*8 FUNCTION COND_LOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C, the functions have the following declarations:

```
int lock_gate(gate_t *gate_p);
long long lock_gate_8(gate8_t *gate_p);
int cond_lock_gate(gate_t *gate_p);
long long cond_lock_gate_8(gate8_t *gate_p);
```

Where *gate\_p* is a pointer of the indicated type.

### Unlocking functions

These functions release a gate from exclusive access. Gates are typically released by the thread that locks them, unless a gate was locked by thread 0 in serial code, in which case it might be unlocked by a single different thread in a parallel construct.

The Fortran gate unlocking functions have the following declarations:

```
INTEGER FUNCTION UNLOCK_GATE(gate)
INTEGER*8 FUNCTION UNLOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C, the functions have the following declarations:

```
int unlock_gate(gate_t *gate_p);
long long unlock_gate_8(gate8_t *gate_p);
```

Where *gate\_p* is a pointer of the indicated type.

### Wait functions

These functions use a barrier to cause the calling thread to wait until the specified number of threads call the function, at which point all threads are released from the function simultaneously.

The Fortran barrier wait functions have the following declarations:

```
INTEGER FUNCTION WAIT_BARRIER(barrier, nthr)
INTEGER*8 FUNCTION WAIT_BARRIER_8(barrier, nthr)
```

Where *barrier* is a *barrier* variable of the indicated type and *nthr* is the number of threads calling the routine.

In C, the functions have the following declarations:

```
int wait_barrier(barrier_t *barrier_p, const int *nthr);
long long wait_barrier_8(barrier8_t *barrier_p, const long long *nthr);
```

Where *barrier\_p* is a pointer of the indicated type and *nthr* is a pointer referencing the number of threads calling the routine.

A barrier variable can be used in multiple calls to the wait function, as long as the programmer ensures that two such barriers are not simultaneously active. It is also the programmer's responsibility to ensure that *nthr* reflects the correct number of threads.

### **sync\_routine directive and pragma**

Among the most basic optimizations performed by the Convex SPP Series compiler is code motion, which is described in Chapter 3, "Compiler optimizations." This optimization can move some code across function calls. If the function call is to a synchronization function that the compiler cannot identify as such, and the code moved must execute on a certain side of it, this movement can cause wrong answers.

The compiler is aware of all synchronization functions presented in this chapter and in Chapter 4, "Basic shared memory programming," and will not move code across them when they appear directly in code. However, if the synchronization function is hidden in a user-defined function, the compiler has no way of knowing about it and may move code across it.

Anytime you call synchronization functions indirectly via your own functions (or directly via CPSlib), you must identify your functions with a `sync_routine` directive or pragma.

In Fortran, `sync_routine` has the following form:

```
C$DIR SPP SYNC_ROUTINE (funclist)
```

In C, it has the following form:

```
#pragma _CNX SPP sync_routine (funclist)
```

Where *funclist* is a comma-delimited list of synchronization procedures.

`sync_routine` is only effective for the listed procedures which lexically follow it in the procedure in which it appears.

Consider the following Fortran example:

```

      INTEGER MY_LOCK, MY_UNLOCK
C$DIR GATE (LOCK)
C$DIR SYNC_ROUTINE (MY_LOCK, MY_UNLOCK)
      .
      .
      .
      LCK = ALLOC_GATE (LOCK)
C$DIR LOOP_PARALLEL
      DO I = 1, N
          LCK = MY_LOCK (LOCK)
          .
          .
          .
          SUM = SUM + A (I)
          LCK = MY_UNLOCK (LOCK)
      ENDDO
      .
      .
      .
      INTEGER FUNCTION MY_LOCK (LOCK)
C$DIR GATE (LOCK)
      LCK = LOCK_GATE (LOCK)
      MY_LOCK = LCK
      RETURN
      END

      INTEGER FUNCTION MY_UNLOCK (LOCK)
C$DIR GATE (LOCK)
      LCK = UNLOCK_GATE (LOCK)
      MY_UNLOCK = LCK
      RETURN
      END

```

Here, `MY_LOCK` and `MY_UNLOCK` are user functions that call the `LOCK_GATE` and `UNLOCK_GATE` intrinsics. The `SYNC_ROUTINE` directive prevents the compiler from moving code across the calls to `MY_LOCK` and `MY_UNLOCK`.

Such a programming technique might be used to implement code that is portable across several parallel architectures which support critical sections using different syntax; `MY_LOCK` and `MY_UNLOCK` could simply be modified to call the correct locking and unlocking functions.

An analogous C example follows:

```
#include <spp_prog_model.h>
main() {
    int i, lck;
    gate_t lock;
#pragma _CNX sync_routine(mylock, myunlock)
    .
    .
    .
    lck = alloc_gate(&lock);
#pragma _CNX loop_parallel(ivar=i)
    for(i=0; i<n; i++) {
        lck = mylock(&lock);
        .
        .
        .
        sum = sum+a[i];
        lck = myunlock(&lock);
    }
}

int mylock(gate_t *lock) {
    int lck;
    lck = lock_gate(lock);
    return lck;
}

int myunlock(gate_t *lock) {
    int lck;
    lck = unlock_gate(lock);
    return lck;
}
```

`sync_routine` is also useful when CPSlib routines are used for synchronization. Refer to Appendix D for more information.

---

### **loop\_parallel(ordered)**

The `loop_parallel(ordered)` directive and `pragma` was briefly introduced in Chapter 4, "Basic shared memory programming." It is designed to be used with ordered sections, which are discussed in the next section, to execute loops with ordered dependencies in loop order. It accomplishes this by parallelizing the loop so that consecutive iterations are initiated on separate processors, in loop order. While `loop_parallel(ordered)` guarantees starting order, it does not guarantee ending order, and it provides no automatic synchronization. To avoid wrong answers, you *must* manually

synchronize dependencies using the ordered section directives, pragmas, or the synchronization intrinsics.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
  DO I = 1, 100
    .
    . !CODE CONTAINING ORDERED SECTION
    .
  ENDDO
```

Or the analogous C code:

```
#pragma _CNX loop_parallel(ordered, ivar = i)
for(i=0;i<100;i++) {
  .
  . /* code containing ordered section */
  .
}
```

Assume that the body of this loop contains code that is parallelizable except for an ordered data dependency (otherwise there is no need to order the parallelization). This dependency is isolated using directives described in the next section. Also assume that 8 threads, numbered 0..7, are available to run the loop in parallel. Each thread would then execute code equivalent to the following:

```
DO I = (my_thread()+1), 100, num_threads()
  ...
ENDDO
```

As shown in Figure 22.

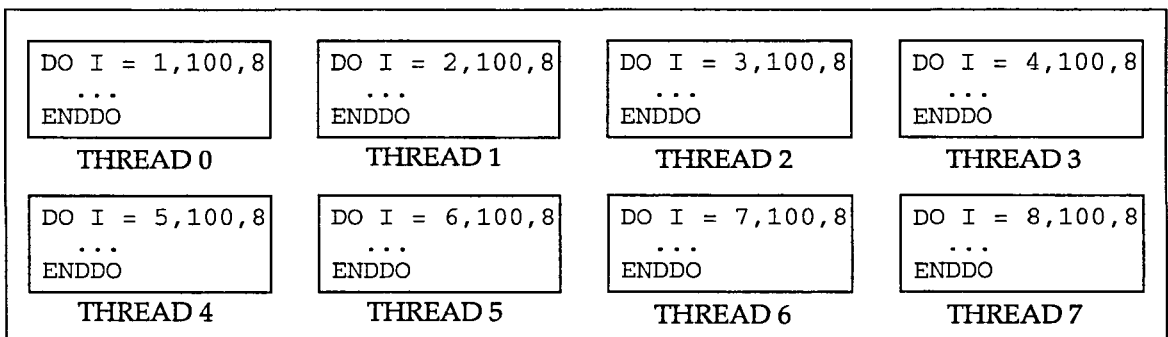


Figure 22 Ordered parallelization

Here, thread 0 executes first, followed by thread 1, and so on; each thread starts its iteration after the preceding iteration has started. A manually-defined ordered section will prevent one thread from executing the code in the ordered section until the previous thread

exits the section, so thread 0 cannot enter the section for iteration 9 until thread 7 exits it for iteration 8. Obviously, this is only efficient if the loop body contains enough code to keep a thread busy until all other threads start their consecutive iterations, thus taking advantage of parallelism. You may find the `max_threads` attribute helpful when fine-tuning `loop_parallel(ordered)` loops to fully exploit their parallel code.

Examples of synchronizing `loop_parallel(ordered)` loops are given in the “Synchronizing code” section.

---

## Critical and ordered sections

As discussed in Chapter 4, “Basic shared memory programming,” critical sections allow you to synchronize simple, nonordered dependencies.

The `critical_section` and `end_critical_section` directives and pragmas are used to specify critical sections. In Fortran, these directives have the following form:

```
C$DIR CRITICAL_SECTION[ (gate) ]  
    ...  
C$DIR END_CRITICAL_SECTION
```

In C, these pragmas have the following form:

```
#pragma _CNX critical_section[ (gate) ]  
    ...  
#pragma _CNX end_critical_section
```

Where *gate* is an optional gate variable used for access to the critical section. *gate* must be appropriately declared as described in the “Gates and barriers” section of this chapter.

The gate variable is required when synchronizing access to a shared variable from multiple parallel tasks. When a gate variable is specified, it must be allocated using the `alloc_gate` intrinsic and should be initialized using the `unlock_gate` intrinsic outside of parallel code prior to use. If no gate is specified, the compiler creates a unique gate for the critical section. When a gate is no longer needed, it should be deallocated using the `free_gate` function.

Critical sections must be entered through the `critical_section` and exited through the `end_critical_section` directive or pragma. They must not contain branches to outside the section. The two directives must appear in the same procedure, but they do not have to be in the same procedure as the parallel construct in which they are used;

i.e. the directives can exist in a procedure which is called in parallel.

Ordered sections, discussed in detail here for the first time, allow you to synchronize dependencies that must execute in iteration order.

The `ordered_section` and `end_ordered_section` directives and pragmas are used to specify critical sections within manually-defined, ordered `loop_parallel` loops only. In Fortran, these directives have the following form:

```
C$DIR ORDERED_SECTION(gate)
...
C$DIR END_ORDERED_SECTION
```

In C, these pragmas have the following form:

```
#pragma _CNX ordered_section(gate)
...
#pragma _CNX end_ordered_section
```

Where *gate* is a required gate variable that must be allocated and, if necessary, unlocked prior to invocation of the parallel loop containing the ordered section. *gate* must be appropriately declared as described in the "Gates and barriers" section of this chapter.

Ordered sections must be entered through the `ordered_section` and exited through the `end_ordered_section` directive or pragma; they cannot contain branches to outside the section. Ordered sections are subject to the same control flow rules as critical sections.

Use critical and ordered sections with care, as they add synchronization overhead to your program. They should only be used when the amount of parallel code is significantly larger than the amount of code containing the dependency.

---

## Synchronizing code

Code containing dependencies can be parallelized by synchronizing the way the parallel tasks access the dependency. This can be done manually using the gates, barriers and synchronization functions, or semiautomatically using critical and ordered sections.

---

### Critical sections

The critical section example on page 125 isolates a single critical section in a loop, so the `critical_section` directive does not require a gate. In this case, the critical section directives automate

allocation, locking, unlocking and deallocation of the needed gate. Multiple dependencies and dependencies in manually-defined parallel tasks can be handled when user-defined gates are used with the directives.

Consider the following Fortran example:

```
      REAL GLOBAL_SUM
C$DIR FAR_SHARED (GLOBAL_SUM)
C$DIR GATE (SUM_GATE)
      .
      .
      .
      LOCK = ALLOC_GATE (SUM_GATE)
C$DIR BEGIN_TASKS
      CONTRIBUT1 = 0.0
      DO J = 1, M
          CONTRIBUT1 = CONTRIBUT1 + FUNC1 (J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIBUT1
C$DIR END_CRITICAL_SECTION
      .
      .
      .
C$DIR NEXT_TASK
      CONTRIBUT2 = 0.0
      DO I = 1, N
          CONTRIBUT2 = CONTRIBUT2 + FUNC2 (J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIBUT2
C$DIR END_CRITICAL_SECTION
      .
      .
      .
C$DIR END_TASKS
      LOCK = FREE_GATE (SUM_GATE)
```

Here, both parallel tasks must access the shared GLOBAL\_SUM variable, which is assigned a function of itself. To ensure that GLOBAL\_SUM is only updated by one task at a time, it is placed in a critical section. The critical sections both reference the

SUM\_GATE variable; this variable is unlocked on entry into the parallel code (gates are always unlocked when they are allocated). When one task reaches the critical section, the CRITICAL\_SECTION directive automatically locks SUM\_GATE. The END\_CRITICAL\_SECTION directive unlocks SUM\_GATE on exit from the section. Since access to both critical sections is controlled by a single gate, the sections must execute one at a time.

An analogous C example follows:

```
static far_shared float global_sum;
static gate_t sum_gate;
.
.
lock = alloc_gate(&sum_gate);
#pragma _CNX begin_tasks
contrib1 = 0.0;
for(j=0;j<m;j++)
    contrib1 = contrib1 + func1(j);
.
.
#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib1;
#pragma _CNX end_critical_section
.
.
#pragma _CNX next_task
contrib2 = 0.0;
for(i=0;i<n;i++)
    contrib2 = contrib2 + func2(j);
.
.
#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib2;
#pragma _CNX end_critical_section
.
.
#pragma _CNX end_tasks
lock = free_gate(&sum_gate);
```

Gated critical sections are also useful in loops containing multiple critical sections, when there are dependencies between the critical sections. If no dependencies exist between the sections, gates are

not needed, as the compiler will automatically supply a unique gate for every critical section lacking a gate.

Consider the following Fortran example:

```
      REAL ABSUM
C$DIR FAR_SHARED(ABSUM)
C$DIR GATE(GATE1)
      LOGICAL ADJB(...)
      .
      .
      .
      LOCK = ALLOC_GATE(GATE1)
C$DIR LOOP_PARALLEL
      DO I = 1, N
          A(I) = B(I) + C(I)
C$DIR  CRITICAL_SECTION(GATE1)
          ABSUM = ABSUM + A(I)
C$DIR  END_CRITICAL_SECTION
          IF(ADJB(I)) THEN
              B(I) = C(I) + D(I)
C$DIR  CRITICAL_SECTION(GATE1)
          ABSUM = ABSUM + B(I)
C$DIR  END_CRITICAL_SECTION
          ENDIF
      .
      .
      .
      ENDDO
      LOCK = FREE_GATE(GATE1)
```

Here, the shared variable `ABSUM` must be updated after `A(I)` is assigned and again if `B(I)` is assigned. Access to `ABSUM` must be guarded by the same gate to ensure that two threads do not attempt to update it at once. The critical sections protecting the assignment to `ABSUM` must explicitly name this gate, or the compiler will choose unique gates for each section, potentially resulting in incorrect answers. Note that there must be a substantial amount of parallelizable code outside of these critical sections to make parallelizing this loop cost-effective.

An analogous C example follows:

```
static far_shared float absum;
static gate_t gate1;
int adjb[...];
.
.
.
lock = alloc_gate(&gate1);
#pragma _CNX loop_parallel(ivar = i)
for(i=0;i<n;i++) {
    a[i] = b[i] + c[i];
#pragma _CNX critical_section(gate1)
    absum = absum + a[i];
#pragma _CNX end_critical_section
    if(adjb[i]) {
        b[i] = c[i] + d[i];
#pragma _CNX critical_section(gate1)
        absum = absum + b[i];
#pragma _CNX end_critical_section
    }
    .
    .
    .
}
lock = free_gate(&gate1);
```

---

## Ordered sections

Like critical sections, ordered sections do the work of locking and unlocking a specified gate to isolate a section of code in a loop. However, they also ensure that the enclosed section of code executes in the same order as the iterations of the ordered parallel loop that contains it. Once a given thread passes through an ordered section, it cannot enter again until all other threads have passed through in order. This ordering is difficult to implement without using the ordered section directives or pragmas.

## Note

**You must use a `loop_parallel(ordered)` directive or pragma to parallelize any loop containing an ordered section.**

Consider the following Fortran code, which contains a backward loop carried dependence on the array A that would normally inhibit parallelization.

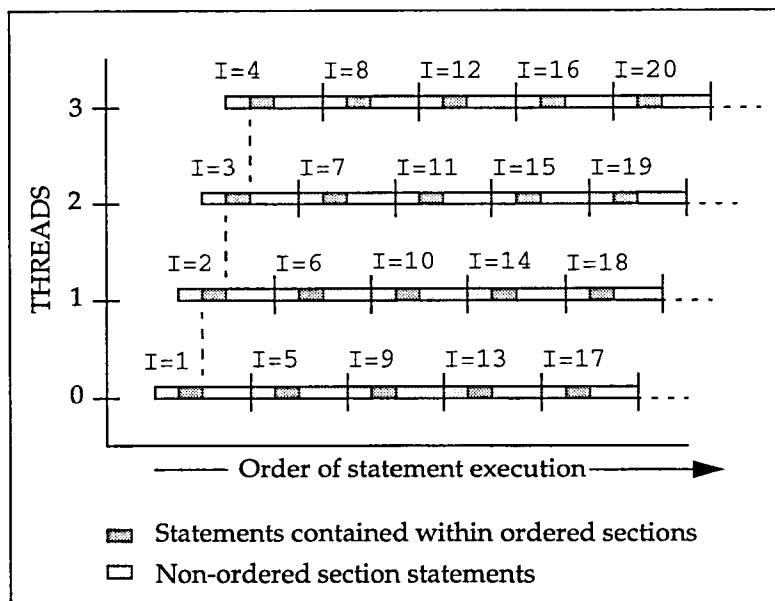
```
DO I = 2, N
  . ! PARALLELIZABLE CODE...
  .
  .
  A(I) = A(I-1) + B(I)
  . ! MORE PARALLELIZABLE CODE...
  .
  .
ENDDO
```

To simplify illustration, we will stick to Fortran, but an analogous C example could similarly be parallelized.

Assuming that the dependence shown is the only one in the loop, and that a significant amount of parallel code exists elsewhere in the loop, we can isolate the dependence and parallelize the loop as shown in the following example:

```
C$DIR GATE(LCD)
      LOCK = ALLOC_GATE(LCD)
      .
      .
      .
      LOCK = UNLOCK_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, N
        . ! PARALLELIZABLE CODE...
        .
        .
C$DIR   ORDERED_SECTION(LCD)
        A(I) = A(I-1) + B(I)
C$DIR   END_ORDERED_SECTION
        . ! MORE PARALLELIZABLE CODE...
        .
        .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

The loop is now parallelized in the manner described in the `loop_parallel(ordered)` section on page 191, and the ordered section containing the `A(I)` assignment will execute in iteration order, ensuring that the value of `A(I-1)` used in the assignment is always valid. Assuming this loop runs on 4 threads, the synchronization of statement execution between threads is illustrated in Figure 23.



**Figure 23** LOOP\_PARALLEL (ORDERED) synchronization

As shown by the dashed lines between initial iterations for each thread, one ordered section must be done before the next is allowed to begin execution. Once a thread exits an ordered section, it cannot reenter it until all other threads have passed through in sequence. Overlap of nonordered statements, represented as lightly shaded boxes, allows all threads to proceed fully loaded, with only brief idle periods on 1, 2, and 3 at the beginning of the loop, and on 0, 1, and 2 at the end.

### Limitations

Each thread in a parallel loop containing an ordered section must pass through the ordered section once and only once on every iteration of the loop. If you execute an ordered section conditionally, you must execute it in all possible branches of the condition; if the code contained in the section is not valid for some branches, you can insert a blank ordered section, as shown in the following Fortran example:

```

C$DIR GATE (LCD)
.
.
.
LOCK = ALLOC_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
DO I = 1, N
.
.
.
IF (Z(I) .GT. 0.0) THEN
C$DIR   ORDERED_SECTION(LCD)
C       HERE'S THE BACKWARD LCD:
        A(I) = A(I-1) + B(I)
C$DIR   END_ORDERED_SECTION
ELSE
C       HERE'S THE BLANK ORDERED SECTION:
C$DIR   ORDERED_SECTION(LCD)
C$DIR   END_ORDERED_SECTION
ENDIF
.
.
.
ENDDO
LOCK = FREE_GATE(LCD)

```

Here, no matter which path through the IF statement the loop takes, it must pass through the ordered section, even though the ELSE section is empty. This allows the compiler to properly synchronize the ordered loop. Note that again, we assume a substantial amount of parallel code exists outside the ordered sections, to offset the synchronization overhead.

An analogous C example follows:

```
static gate_t lcd;
.
.
.
lock = alloc_gate(&lcd);
#pragma _CNX loop_parallel(ordered,ivar = i)
for(i=0;i<ni++) {
    .
    .
    .
    if(z[i] > 0.0) {
#pragma _CNX ordered_section(lcd)
        a[i] = a[i-1] + b[i]; /* backward lcd */
#pragma _CNX end_ordered_section
    } else {
#pragma _CNX ordered_section(lcd)
        /* here's the blank ordered section */
#pragma _CNX end_ordered_section
    }
    .
    .
    .
}
lock = free_gate(&lcd);
```

Ordered sections within nested loops can create similar, but more difficult to recognize, problems. Consider the following Fortran example (gate manipulation is omitted for brevity):

```
C$DIR LOOP_PARALLEL(ORDERED)
    DO I = 1, 99
        DO J = 1,M
            .
            .
            .
C$DIR    ORDERED_SECTION(ORDGATE)
        A(I,J) = A(I+1,J)
C$DIR    END_ORDERED_SECTION
            .
            .
            .
        ENDDO
    ENDDO
```

Recall that once a given thread has passed through an ordered section, it cannot reenter it until all other threads have passed through in order. This is only possible in the given example if the

number of available threads integrally divides 99 (the  $I$  loop limit). If not, deadlock results.

To see why, assume 6 threads, numbered 0 through 5, are running the parallel  $I$  loop. For  $I = 1, J = 1$ , thread 0 passes through the ordered section and loops back through  $J$ , stopping when it reaches the ordered section again for  $I = 1, J = 2$ . It can't enter until threads 1 through 5 (which are executing  $I = 2$  through 6,  $J = 1$  respectively) pass through in sequence. This is not a problem, and the loop proceeds through  $I = 96$  in this fashion in parallel. However, for  $I > 96$ , all 6 threads are no longer needed. In a single loop nest this would not pose a problem; the leftover 3 iterations would be handled by threads 0 through 2; when thread 2 exited the ordered section it would hit the ENDDO and the  $I$  loop would terminate normally. But in this example, the  $J$  loop isolates the ordered section from the  $I$  loop, so thread 0 executes  $J = 1$  for  $I = 97$ , loops through  $J$  and waits during  $J = 2$  at the ordered section for thread 5, which has gone idle, to complete. Threads 1 and 2 similarly execute  $J = 1$  for  $I = 98$  and  $I = 99$ , and similarly wait after incrementing  $J$  to 2. The entire  $J$  loop must terminate before the  $I$  loop can terminate, but the  $J$  loop can never terminate because the idle threads 3, 4, and 5 never pass through the ordered section. Deadlock results.

The analogous C code looks like this:

```
#pragma _CNX loop_parallel(ordered, ivar = i)
for(i=0; i<99; i++) {
    for(j=0; j<m; j++) {
        .
        .
        .
    #pragma _CNX ordered_section(ordgate)
        a[i][j] = a[i+1][j];
    #pragma _CNX end_ordered_section
        .
        .
        .
    }
}
```

To handle this problem, you can expand the ordered section to include the entire J loop, as shown in the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 99
C$DIR  ORDERED_SECTION(ORDGATE)
      DO J = 1,M
          .
          .
          .
          A(I,J) = A(I+1,J)
          .
          .
          .
      ENDDO
C$DIR  END_ORDERED_SECTION
      ENDDO
```

In this approach, each thread executes the entire J loop each time it enters the ordered section, allowing the I loop to terminate normally regardless of the number of threads available.

The analogous C code follows:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(i=0;i<99;i++) {
#pragma _CNX ordered_section(ordgate)
    for(j=0;j<m;j++) {
        .
        .
        .
        a[i][j] = a[i+1][j];
        .
        .
        .
    }
#pragma _CNX end_ordered_section
}
```

Another approach is to manually interchange the I and J loops, as shown in the following example:

```
        DO J = 1, M
C$DIR   LOOP_PARALLEL(ORDERED)
        DO I = 1, 99
            .
            .
            .
C$DIR   ORDERED_SECTION(ORDGATE)
        A(I,J) = A(I+1,J)
C$DIR   END_ORDERED_SECTION
            .
            .
            .
        ENDDO
    ENDDO
```

Here, the I loop is parallelized on every iteration of the J loop. Again, the ordered section is not isolated from its parent loop, so the loop can terminate normally. This example has added benefits; here, elements of A are accessed more efficiently, and this nest can be further optimized by parallelizing the J loop across hypernodes by using a LOOP\_PARALLEL(NODES) directive.

The analogous C code follows:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(j=0;j<99;j++) {
    for(i=0;i<m;i++) {
        .
        .
        .
#pragma _CNX ordered_section(ordgate)
        a[i][j] = a[i+1][j];
#pragma _CNX end_ordered_section
        .
        .
        .
    }
}
```

---

## Manual synchronization

Ordered and critical sections allow you to isolate dependencies in a structured, semiautomatic manner. The same isolation can be accomplished manually using the functions discussed in the “Synchronization functions” section on page 185.

Recall the simple critical section example presented on page 125:

```
C$DIR LOOP_PARALLEL
      DO I = 1, N ! LOOP IS PARALLELIZABLE
        .
        .
        .
C$DIR  CRITICAL_SECTION
      SUM = SUM + X(I)
C$DIR  END_CRITICAL_SECTION
        .
        .
        .
      ENDDO
```

As shown, this example is easily implemented using critical sections. It can be manually implemented in Fortran as shown below.

```
C$DIR GATE(CRITSEC)
      .
      .
      .
      LOCK = ALLOC_GATE(CRITSEC)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        LOCK = LOCK_GATE(CRITSEC)
        SUM = SUM + X(I)
        LOCK = UNLOCK_GATE(CRITSEC)
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(CRITSEC)
```

As shown, the manual implementation requires declaring, allocating and deallocating a gate, which must be locked on entry into the critical section using the `LOCK_GATE` function and unlocked on exit using `UNLOCK_GATE`.

An analogous manual C implementation follows:

```
static gate_t critsec;
.
.
.
lock = alloc_gate(&critsec);
#pragma _CNX loop_parallel(ivar = i)
for(i=0;i<n;i++) {
.
.
.
lock = lock_gate(&critsec);
sum = sum + x[i];
lock = unlock_gate(&critsec);
.
.
.
}
lock = free_gate(&critsec);
```

Using synchronization functions to implement critical and ordered sections generally requires more work, and the compiler will not check such constructs for errors as thoroughly as it will check directive-delimited sections. However, because the functions are unstructured, they can be used to create more complex critical regions than are supported by the directives.

Consider the following Fortran example:

```
C$DIR GATE(TASK1, TASK2)
.
.
.
LOK1 = ALLOC_GATE(TASK1)
LOK2 = ALLOC_GATE(TASK2)
LOK1 = LOCK_GATE (TASK1) ! LOCKING HERE PREVENTS 3RD TASK
LOK2 = LOCK_GATE (TASK2) ! FROM ACCESSING A OR B BEFORE
! FIRST TWO TASKS ASSIGN THEM
C$DIR BEGIN_TASKS(ORDERED) ! TASK ONE:
DO I = 1, N
A(I) = 2.0 * A(I) + C(I) ! A GETS ASSIGNED IN THIS TASK
ENDDO
LOK1 = UNLOCK_GATE (TASK1) ! A CAN NOW BE ACCESSED
C$DIR NEXT_TASK ! TASK TWO:
DO J = 1, N
B(J) = C(J) * SIN (X(J)) ! B GETS ASSIGNED IN THIS TASK
ENDDO
LOK2 = UNLOCK_GATE (TASK2) ! B CAN NOW BE ACCESSED
```

```

C$DIR NEXT_TASK      ! TASK THREE:
DO K = 1, N          ! COMPUTE P IN PARALLEL WITH A AND B:
  P(K) = EXP (3.0 * SQRT (Q(K))) / ATAN (R(K))
ENDDO
LOK1 = LOCK_GATE (TASK1)      ! WAIT FOR UNLOCK IN TASK 1
LOK1 = UNLOCK_GATE (TASK1)    ! WHEN LOCK IS ATTAINED, UNLOCK
LOK2 = LOCK_GATE (TASK2)      ! WAIT FOR UNLOCK IN TASK 2
LOK2 = UNLOCK_GATE (TASK2)    ! WHEN ATTAINED, UNLOCK
DO L = 1, N                ! WHEN WE HAVE BOTH LOCKS,
  D(L) = P(L) * B(L) / A(L) ! COMPUTE D
ENDDO
C$DIR NEXT_TASK      ! TASK FOUR:
DO M = 1, N          ! NO DEPENDENCIES IN THIS TASK
  Y(M) = X(M) * Y(M) / Z(M) ! Y CAN BE COMPUTED WITH
ENDDO                ! A, B, P, D
C$DIR END_TASKS
LOK1 = FREE_GATE (TASK1)
LOK2 = FREE_GATE (TASK2)

```

Here, the `BEGIN_TASKS (ORDERED)` directive guarantees that the following parallel tasks begin in lexical order (ending order is indeterminate). Ordered sections, however, cannot be used with parallel tasks. To order the dependency in task 3 of this code, where the computation of `D` assumes that `A`, `B` and `P` are fully computed, we must use manually locked and unlocked gates.

Tasks 1, 2, 4 and the `K` loop of task 3 can all run in parallel. To allow this while postponing execution of the `L` loop in task 3 until `A`, `B` and `P` are computed, we allocate and lock two gates, named `TASK1` and `TASK2`, before the parallel tasks begin. `TASK1` remains locked until `A` is computed, at which point it is unlocked; `TASK2` remains locked until `B` is computed, and is then unlocked. The `I` and `J` loops are free to run in parallel, along with the `K` loop in task 3 and the `M` loop in task 4, since none of these loops depend on each other's gates. The `L` loop in task 3, however, cannot begin execution until task 3 can lock both `TASK1` and `TASK2` gates. Task 3 immediately unlocks these gates because it doesn't need them; their purpose was to force it to wait until `A`, `B` and `P` were computed. Once task 3 has acquired and relinquished both locks, the `L` loop is free to run.

A similar C example follows:

```
static gate_t task1, task2;
.
.
.
lok1 = alloc_gate(&task1);
lok2 = alloc_gate(&task2);
lok1 = lock_gate(&task1); /* locking here prevents 3rd task */
lok2 = lock_gate(&task2); /* from accessing a or b before */
                          /* first two tasks assign them */
#pragma _CNX begin_tasks (ordered) /* task 1: */
for(i=0;i<n;i++)
    a[i] = 2.0 * a[i] + c[i]; /* a gets assigned in this task */
lok1 = unlock_gate(&task1); /* a can now be accessed */
#pragma _CNX next_task          /* task 2: */
for(j=0;j<n;j++)
    b[j] = c[j] * sin(x[j]); /* b gets assigned in this task */
lok2 = unlock_gate(task2); /* b can now be accessed */
#pragma _CNX next_task          /* task 3: */
for(k=0;k<n;k++) /* compute p in parallel with a and b */
    p[k] = exp(3.0*sqrt(q[k]))/atan(r[k]);
lok1 = lock_gate(&task1); /* wait for unlock in task 1 */
lok1 = unlock_gate(&task1); /* when lock is attained, unlock */
lok2 = lock_gate(&task2); /* wait for unlock in task 2 */
lok2 = unlock_gate(&task2); /* when attained, unlock */
for(l=0;l<n;l++) /* when we have both locks, */
    d[l] = p[l] * b[l]/a[l]; /* compute d */
#pragma _CNX next_task          /* task 4: */
for(m=0;m<n;m++) /* no dependencies in this task */
    y[m] = x[m] * y[m]/z[m]; /* y can be computed with a,b,p,d */
#pragma _CNX end_tasks
lok1 = free_gate(&task1);
lok2 = free_gate(&task2);
```

Another advantage of manually-defined critical sections is the ability to conditionally lock them. This allows the task that wishes to execute the section to proceed with other work if the lock cannot be acquired. This construct is useful, for example, in situations where one thread is performing I/O for several other parallel threads; while a processing thread is reading from the input queue, the queue is locked, and the I/O thread can move on to do output. While a processing thread is writing to the output queue, the I/O thread can do input. This allows the I/O thread to keep as busy as possible while the parallel computational threads execute their (presumably large) computational code.

## Advanced shared memory example

We will now consider a more detailed Fortran example which is manually parallelized in two dimensions and makes use of barrier synchronization and `block_shared` arrays, two topics which are difficult to illustrate in brief examples.

Consider the following Fortran code:

```

REAL*8 A(800,800,800), B(800,800,800)
.
.
.
DO K = 1,800
  DO J = 1, 800
    DO I = 1, 800
      B(I,J,K) = ...
      .
      .
      .
    ENDDO
  ENDDO
ENDDO
.
.
.
DO K = 2,799
  DO J = 2, 799
    DO I = 2, 799
      A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
>      B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
      .
      .
      .
    ENDDO
  ENDDO
ENDDO

```

While this code contains several opportunities for parallelization, each of its arrays occupies nearly 4 Gbytes. The two arrays together with any additional arrays or variables required by the program therefore require far more virtual memory than is available to the process.

To get around this, we must somehow increase the virtual address space available to the process. We can do this by requiring that the process run on a minimum number of hypernodes, and allocating sections of A and B in `node_private` memory on the individual hypernodes which will compute them. Since these sections will only be accessible to the hypernodes on which they reside, any elements that must be shared must be copied into shared memory. We will use `block_shared` memory for this function. Recall that

block\_shared arrays are distributed in chunks across hypernodes, allowing fastest access from the hypernode on which the section resides (like node\_private), but also allowing other hypernodes to access the data. This will provide optimal array-access efficiency.

The following implementation uses manually-partitioned arrays and two-dimensional manual parallelization to solve this example as efficiently as possible. It assumes that performance testing indicates that a minimum of 4 hypernodes is required to adequately provide the necessary amounts of virtual memory.

```

REAL*8 A, B
C EACH NODE WILL ALLOCATE A 1/NUM_NODES-SIZE SECTION OF EACH ARRAY:
  ALLOCATABLE A(:,:,:),B(:,:,:)
C$DIR NODE_PRIVATE(A,B)
C   REQUIRED FOR DYNAMIC NODE_PRIVATE ALLOCATED BY A SINGLE THRD:
C$DIR FAR_SHARED_POINTER(A,B)
  REAL*8 BLEFT, BRIGHT ! EACH NODE COMMUNICATES ITS ENDPANES
  ALLOCATABLE BLEFT(:,:,:),BRIGHT(:,:,:) ! TO OTHER NODES VIA
C$DIR BLOCK_SHARED(BLEFT, BRIGHT)          ! BLEFT/BRIGHT
  INTEGER NODE_ID, nodeNS      ! EACH NODE'S ID AND ARRAY SIZE
  INTEGER K, J, I, KS, KE      ! INDUCTION VARIABLES
C$DIR NODE_PRIVATE(NODE_ID, nodeNS, KS, KE)
C$DIR THREAD_PRIVATE(K, J, I, LBAR2)
C$DIR BARRIER(BARRALL) ! TO PREVENT USING B BEFORE IT IS ASSIGNED
  INTEGER CEILING
  CEILING(X) = AINT(1.0+X) - AINT(1.0+AINT(X) - X)
  .
  .
  .
PRINT*, "ENTER NUMBER OF HYPERNODES:"
READ*, NN
IF (NN .GE. 4) THEN          ! ENOUGH MEMORY TO SOLVE
C   NS IS THE MAX ARRAY SIZE NEEDED BY AT LEAST 1 NODE:
  NS = CEILING(800./FLOAT(NN)) ! SIZE OF A,B ARRAY SECTIONS
ELSE
  STOP 'NOT ENOUGH MEMORY' ! STOP IF NOT ENOUGH MEMORY
ENDIF
  .
  .
  .
C   ALLOCATE EACH NODE'S ARRAYS;
C   B'S 3RD DIMENSION HAS ROOM FOR NEIGHBOR VALUES:
  ALLOCATE(A(800,800,NS),B(800,800,0:NS+1))
C   ALLOCATE BLOCK_SHARED BLEFT AND BRIGHT TO ALLOW EACH NODE
C   TO WRITE ENDPANES LOCALLY, SHARE NEIGHBORS GLOBALLY:
  ALLOCATE(BLEFT(800,800,NN),BRIGHT(800,800,NN))
  LBAR1 = ALLOC_BARRIER(BARRALL) ! ALLOCATE BARRIER

```

```

C$DIR LOOP_PARALLEL(NODES)
  DO NODE = 1, NN          ! BEGIN NODE-PARALLEL STRUCTURE:
    NODE_ID=MY_NODE()+1   ! INDEX INTO SHARED ARRAY.
  C    FIND EXACT SIZE OF ARRAYS TO BE COMPUTED ON EACH NODE:
    IF (NODE_ID .GT. 800 - NN * (NS-1)) THEN
      nodeNS=NS-1 ! HIGH NUMBERED NODES MIGHT DO ONE LESS.
    ELSE
      nodeNS = NS ! ALL OTHER NODES DO THE SAME SIZE.
    ENDIF
  C    LOOP NEST 1:
C$DIR LOOP_PARALLEL(THREADS) !IN THRD-PARALLEL LOOP, EACH NODE
  DO K=1,nodeNS           !COMPUTES ITS PIECE OF THE B ARRAY
    DO J = 1, 800
      DO I = 1, 800
        B(I,J,K) = ...
        .
        .
        .
      ENDDO
    ENDDO
  ENDDO ! END LOOP NEST 1
  C    LOOP NEST 2:
  C    NOW PUT EACH NODE'S B ENDPLANES IN SHARED ARRAYS:
  DO J = 1, 800
    DO I = 1, 800
  C      CURRENT NODE'S RIGHT PLANE IS NEIGHBOR NODE'S LEFT:
      BLEFT(I,J,NODE_ID)=B(I,J,nodeNS)
  C      CURRENT NODE'S LEFT IS NEIGHBOR NODE'S RIGHT:
      BRIGHT(I,J,NODE_ID)=B(I,J,1)
    ENDDO
  ENDDO ! END LOOP NEST 2
  C    BARRIER ENSURES ALL ENDPLANES ARE COPIED BEFORE USED:
  LBAR2=WAIT_BARRIER(BARRALL,NN) !ONE THREAD PER NODE CALLS
  C    LOOP NEST 3:
  IF (NODE_ID .GT. 1) THEN !THIS NODE IS INTERIOR OR RTMOST
    DO J = 1, 800
      DO I = 1, 800
        B(I,J,0)=BLEFT(I,J,NODE_ID-1) !GET LEFT ENDPLANE
      ENDDO
    ENDDO
  ENDDO
  ENDIF
  IF (NODE_ID .LT. NN) THEN !THIS NODE IS INTERIOR OR LFTMOST
    DO J = 1, 800
      DO I = 1, 800
        B(I,J,nodeNS+1)=BRIGHT(I,J,NODE_ID+1) !GET RT ENDPLANE
      ENDDO
    ENDDO
  ENDDO
  ENDIF

```

```

C      GOT THE ENDPANES, NOW COMPUTE ARRAY BOUNDS:
      IF (NODE_ID .EQ. 1) THEN ! THIS NODE IS LEFTMOST
          KS = 2
          KE = nodeNS
      ELSEIF (NODE_ID .EQ. NN) THEN ! THIS NODE IS RIGHTMOST
          KS = 1
          KE = nodeNS-1
      ELSE
          ! THIS NODE IS INTERIOR
          KS = 1
          KE = nodeNS
      ENDIF
C      NOW COMPUTE ARRAY A:
C      LOOP NEST 4:
C$DIR  LOOP_PARALLEL(THREADS) ! GO THREAD-PARALLEL ON EACH NODE
      DO K = KS, KE
          DO J = 2, 1023
              DO I = 2, 1023
                  A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
>                B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
                  .
                  .
                  .
              ENDDO
          ENDDO
      ENDDO ! END LOOP NEST 4
LBAR1 = FREE_BARRIER(BARRALL) ! DEALLOCATE BARRIER
ENDDO ! END NODE-PARALLEL LOOP

```

Here, the A and B arrays, which are used in the bulk of the compute-intensive code and occupy the bulk of the process's virtual memory, are assigned the `node_private` class. They are allocated in serial code, so that when they are accessed physical copies are created on each hypernode, and each hypernode accesses its copy using the same array names. `far_shared` pointers are used to access A and B, as described on page 160.

If enough hypernodes aren't available to handle the virtual memory requirements of A and B at allocation time, the program stops.

Note that B is allocated with an extra K dimension plane at each end so that the hypernodes can satisfy computational dependencies involving these endplanes by sharing them.

B is computed in loop nest 1, and is then used in loop nest 4 to compute A. B must be entirely computed before computation of A can begin. Because the computation of A contains both positive and negative offset indexing of B, each hypernode must share its B endplanes using the extra endplanes allocated before computation of A can commence.

This sharing is accomplished through the `BLEFT` and `BRIGHT` `block_shared` arrays. Loop nest 2 copies the endplanes from `B` into these arrays, and, after the `BARRALL` barrier ensures that this copying is complete, loop nest 3 copies the endplanes from `BLEFT` and `BRIGHT` into the appropriate extended dimensions of `B` on the appropriate hypernodes. Note that `BLEFT` and `BRIGHT` are allocated based on the number of hypernodes, and indexed in their third dimension by `NODE_ID`; this ensures that the elements most often used on a hypernode will physically reside on that hypernode. This means copying the endplanes of `B` into these arrays can be done with minimal access latency. Copying the endplanes into neighboring hypernodes' copies of `B` in loop nest 3 requires more costly interhypernode memory accesses. However, if the problem is to run on multiple hypernodes, which it must to obtain the virtual memory it needs, some interhypernode communication is inevitable, and this code minimizes it to the extent possible.

After this copying takes place, each hypernode determines its index range in the third dimension of `A` and `B`, and `A` is computed in loop nest 4. Here, all accesses are to `node_private` memory, so access latency is minimal.

This problem can be solved in parallel in a number of ways, which include splitting `A` and `B` up across threads in `thread_private` memory, or placing `BLEFT` and `BRIGHT` in `far_shared` memory. The solution presented above is more versatile and efficient.

This problem can also be solved using message passing, as explained in Chapter 7, "Message passing programming."



This chapter presents a high level overview of the message passing paradigm, in which all parallelism and data distribution must be explicitly handled by the programmer. Tasks are coordinated and data is shared among processes by passing messages using the functions of the ConvexPVM library, which are accessible from both C and Fortran. This chapter assumes a familiarity with ConvexPVM and is therefore primarily concerned with providing Exemplar-specific PVM coding tips and examples. For more information on using ConvexPVM, refer to the *ConvexPVM Release Notice*. For more information on specific ConvexPVM functions, including code examples, refer to the man page for the function in question.

The message passing paradigm is useful for porting message passing programs written for other systems, or for those programmers already familiar with programming in this style.

A brief comparison of the message passing and shared memory paradigms is given in Chapter 1, "Introduction."

---

## Basic operation

In this programming style, the user writes an application to be run as a collection of single-threaded processes which can be run on one or more processors. Typically each process determines its actions at runtime based on its assigned task identifier (tid), which resembles the UNIX process-identifier (pid) but includes encoded information regarding the location of the process. The tid is available through a ConvexPVM function. The user also specifies the communication of data values among the processes using message passing.

Message passing programs are inherently parallel, and unless explicitly coordinated by message waiting, all processes execute independently. In a conventionally coded message passing

program, all variables are private to each process. So regardless of whether variables have been declared to be in any of the memory classes, no process can access the variables of any other process. Synchronization among the processes occurs explicitly through message passing.

---

## Message passing library

The message passing library is ConvexPVM (Parallel Virtual Machine). ConvexPVM functions are written in C, but are accessible from both C and Fortran programs. The Fortran versions automatically perform appropriate argument translations and use the correct calling conventions for calling C.

---

### Exemplar-specific configuration features

ConvexPVM for SPP Series machines includes extensions to the hostfile syntax, new default system filenames, performance tuning functions, and a new architecture specifier.

#### Convex-specific hostfile syntax

When multiple subcomplexes are used in a virtual machine, they communicate via sockets and must therefore appear in separate entries in the hostfile. Use the following form to specify a subcomplex in the hostfile:

```
myhostname : mysubcomplex
```

Where *myhostname* is the name of the machine and *mysubcomplex* is the name or numeric subcomplex ID of the subcomplex.

#### System filenames

On SPP Series machines, default names for PVM system files include subcomplex IDs. These files use the following form:

```
/tmp/pvmd.uid.scid
```

Where *uid* is the user ID and *scid* is the subcomplex ID of the task which generated the file.

#### Performance tuning functions

ConvexPVM for SPP Series machines includes the `pvm_setopt(int option, int value)`, and `pvmf_setopt(option, value, oldvalue)` performance tuning functions. Refer to the `setopt(3pvm)` man page for more information.

## New architecture specifier

The `pvm_spawn` and `pvmf_spawn` functions now recognize CSPP as the architecture specifier for SPP Series machines.

## PVMFRAGSIZE environment variable

This environment variable is used to specify the size (in 4 kbyte pages) used for shared memory message fragments. The default is 4 pages (16 kbytes).

## Other features

Always consult the *ConvexPVM Release Notice* for your particular version of ConvexPVM for the most up-to-date information on ConvexPVM features.

---

## Exemplar intertask communication

ConvexPVM automatically routes messages between tasks running in one subcomplex on an Exemplar system via shared memory. This greatly decreases message latency over traditional PVM socket-based communication.

Messages to other machines or to other subcomplexes are routed via sockets.

## ConvexPVM buffering on Exemplar

Traditional, distributed-memory networks of machines running ConvexPVM execute the following steps to transfer a message:

1. The sending task packs the message into a send buffer in local memory.
2. The buffer is sent over the network to the receiving task by the ConvexPVM daemon (unless a direct connection exists, in which case the daemon is circumvented), and placed in a receive buffer at the receiving end.
3. The receiving task unpacks the message.

The use of shared memory for intrasubcomplex communication on Exemplar systems greatly speeds this process. Rather than each task allocating send and receive buffers in private memory, the sending task packs its data into a reserved area of shared memory. The receiving task can then unpack the data from shared memory. The ConvexPVM daemon never gets involved, and since shared memory is used to transfer the data, the actual transfer time is negligible compared to network-based message passing. In fact, packing and unpacking message data is the most time-consuming part of message passing on Exemplar systems.

Another advantage of shared memory message buffering is the larger buffer size available on Exemplar. Because the message buffer size is user-definable and limited only by the amount of shared memory available to the subcomplex, potentially many more messages can be pipelined by the sending task, allowing the task to continue with other work while the messages are being unpacked by the receiving tasks. On other systems, limited network bandwidth can sometimes block the sending task from doing parallel work while it waits to send data across the network.

When calling the `pvm_initsend`, `PVMFINITSEND`, `pvm_mkbuf`, or `PVMFMKBUF` functions in preparation for packing a message buffer, it is most efficient on Exemplar systems to specify no encoding in the `encoding` argument. In Fortran, use the constant `PVMRAW` to do this; in C, use `PvmDataRaw`. These values will enhance performance when sending messages between Exemplar and HP systems also.

---

## Message passing approaches to parallelism

ConvexPVM programs generally take one of two approaches to parallelism: the master/slave approach, or the single program multiple data (SPMD) approach.

In the master/slave approach, a set of computational slave processes perform work for one or more master processes. This approach is generally used when little synchronization is required between tasks.

In the more common SPMD approach, the program spawns several identical tasks which perform the same work independently on different data sets. In this approach, synchronization is often required between parallel tasks. Exemplar systems running ConvexPVM are especially suited to this model because of their fast shared memory communication, which minimizes synchronization delays.

The following sections present both master/slave and SPMD examples. In both cases, the number of processors is hard coded at 4. These examples are presented in Fortran to simplify explanation; they can also be implemented in C.

---

## Master/slave example

The example presented here uses the master/slave paradigm to implement the following matrix multiply algorithm:

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

To do this, the master program spawns four slave programs and broadcasts the same column of B to each. The slaves receive this column into a single dimensional array, and each uses it, along with a section of A, to simultaneously compute a disjoint, single-dimensional section of C. These sections of C are passed back to the master as they are computed; when the master receives sections of C from all four slaves, it broadcasts new columns of B back to them, and the process is repeated until all of C has been computed.

The master program follows:

```
PROGRAM MASTER
REAL*4 C(40,40),B(40,40)
INTEGER TIDS(4),NPROCS,N,MY_NUM,J,I
CHARACTER*4 ARCH
CHARACTER*24 SLAVENAME
CHARACTER*24 OUTFILE
INCLUDE '/USR/INCLUDE/FPVM3.H'

N = 40

C ENROLL UNDER PVM:
CALL PVMFMYTID(MY_NUM)
CALL INIT_OUTFILE

NPROCS = 4
ARCH = "CSPP"
SLAVENAME = "SLAVE" !NAME OF SLAVE PROGRAM

C SPAWN SLAVE PROCESSES:
CALL PVMFSPAWN(SLAVENAME,PVMDEFAULT,'*',NPROCS,TIDS,NUMT)
IF(NUMT .LT. NPROCS) THEN ! INDICATES SPAWN FAILURE
  WRITE(6,*) 'PVMFSPAWN FAILURE; NUMT =',NUMT
ENDIF
```

```

C INITIALIZE B ARRAY:
    CALL INIT_ARRAY(B)

C CLEAR DEFAULT SEND BUFFER AND SPECIFY NO MESSAGE ENCODING:
    CALL PVMFINITSEND(PVMRAW,IBUFID)

    MSGTYPE = 1 ! INIT MSG WILL BE RECEIVED BY ALL SLAVES;
                ! FOR MSGTYPE >= 2, EVEN MSGTYPE VALUES
                ! INDICATE RAW DATA BEING SENT/RECEIVED; ODD
                ! VALUES INDICATE RESULTS ARE BEING SENT/RECVD.

C PACK ARRAY DIMENSION SIZE:
    CALL PVMFPACK(INTEGER4,N,1,1,INFO)

C PACK NUMBER OF PROCESSORS:
    CALL PVMFPACK(INTEGER4,NPROCS,1,1,INFO)

C PACK TIDS ARRAY:
    CALL PVMFPACK(INTEGER4,TIDS,NPROCS,1,INFO)

C ALL INFO ON CURRENT MESSAGE BUFFER IS INIT INFO FOR ALL SLAVES;
C MULTICAST MESSAGE BUFFER TO ALL THREADS:
    CALL PVMFMCAST(NPROCS,TIDS,MSGTYPE,INFO)

C LOOP OVER ARRAY COLUMNS:
    DO J = 1,N
        MSGTYPE = MSGTYPE + 1 ! INCREMENT FOR THREAD-SPECIFIC MSGS
        CALL PVMFINITSEND(PVMRAW,IBUFID)

C     PACK COLUMN NUMBER:
        CALL PVMFPACK(INTEGER4,J,1,1,INFO)

C     PACK CURRENT COLUMN OF B:
        CALL PVMFPACK(REAL4,B(1,J),N,1,INFO)

C     MULTICAST ARRAY COL; MSGTYPE HERE IS ALWAYS EVEN, INDICATING
C     RAW DATA:
        CALL PVMFMCAST(NPROCS,TIDS,MSGTYPE,INFO)

        MSGTYPE = MSGTYPE + 1 ! MSGTYPE BECOMES ODD
        NR = N/NPROCS ! NUMBER OF COLS OF C TO RECEIVE BELOW
C LOOP TO RECEIVE RESULTS FROM EVERY PROCESSOR:
        DO I = 1,NPROCS

C             RECEIVE RESULTS FROM SLAVES:
                CALL PVMFREC(-1,MSGTYPE,IBUFID)

C UNPACK ROW NUMBER:
        CALL PVMFUNPACK(INTEGER4,IROW,1,1,INFO)

```

```

C UNPACK COLUMN NUMBER:
    CALL PVMFUNPACK(INTEGER4,ICOL,1,1,INFO)

C UNPACK NR ARRAY DATA ELEMENTS STARTING AT IROW,ICOL:
    CALL PVMFUNPACK(REAL4,C(IROW,ICOL),NR,1,INFO)
ENDDO
ENDDO

CALL OUTPUT(C,N)

CALL PVMFEXIT(INFX)
END

```

Here, the *J* loop loops over columns of *B*, broadcasting them to all slave processes. The *I* loop then loops over the number of slaves, ensuring that all results are received before allowing *J* to continue by broadcasting another column of *B*.

Now consider the slave program:

```

PROGRAM SLAVE
DIMENSION A(10,40)
DIMENSION B(40),CIJ(10)
DIMENSION TIDS(4)
INTEGER MASTERNUM, IBUFID, NUMBYTES
CHARACTER*17 OUTFILE
INCLUDE '/USR/INCLUDE/FPVM3.H'

```

```

C ENROLL IN PVM AND GET TID:
    CALL PVMFMYTID(MY_NUM)

C CREATE OUTPUT FILE
    CALL INIT_OUTFILE

    MSGTYPE = 1 ! SET MSGTYPE TO RECEIVE INITIALIZATION MSG

C ALL SLAVES RECEIVE INITIALIZATION MESSAGE:
    CALL PVMFRECV(-1,MSGTYPE,IBUFID)
    CALL PVMFBUFINFO (IBUFID, NUMBYTES, MSGTYPE, MASTERNUM, INFO)

C UNPACK ARRAY DIMENSION:
    CALL PVMFUNPACK(INTEGER4,N,1,1,INFO)

C UNPACK NUMBER OF PROCESSORS:
    CALL PVMFUNPACK(INTEGER4,NPROCS,1,1,INFO)

C UNPACK TIDS ARRAY:
    CALL PVMFUNPACK(INTEGER4,TIDS,NPROCS,1,INFO)

```

```

C DETERMINE NUMBER OF ROWS RECEIVED:
  NR = N / NPROCS

  CALL INIT_ARRAY(A)

C EACH SLAVE DETERMINES THE ROW NUMBER OF ITS DATA:
  DO I = 1,NPROCS ! LOOP OVER ALL PROCESSORS
    IF(MY_NUM .EQ. TIDS(I)) THEN
      INUM = I - 1 ! SET ROW NUMBER ACCORDING TO TID
    ENDIF
  ENDDO

  IROW = INUM * NR ! DETERMINE UPPER LIMIT FOR ROWS

C LOOP OVER COLUMNS; EACH SLAVE RECEIVES, PROCESSES AND RETURNS DATA
C IN THIS LOOP:
  DO J = 1,N
    MSGTYPE = MSGTYPE + 1 !MSGTYPE BECOMES EVEN FOR RAW DATA

C RECEIVE DATA FROM MASTER:
    CALL PVMFRECVM(MASTERNUM,MSGTYPE,IBUFID)

C UNPACK COLUMN NUMBER:
    CALL PVMFUNPACK(INTEGER4,ICOL,1,1,INFO)

C UNPACK N DATA ELEMENTS OF B; NOTE COLUMNS ARE HANDLED AS
C SINGLE-DIMENSIONAL ARRAYS IN SLAVE PROCESSES:
    CALL PVMFUNPACK(REAL4,B(1),N,1,INFO)

C PERFORM MATRIX MULTIPLY:
    DO I = 1,NR
      CIJ(I) = 0.0
    ENDDO
    DO K = 1,N
      DO I = 1,NR
        CIJ(I) = CIJ(I) + A(I,K) * B(K)
      ENDDO
    ENDDO

C RETURN RESULTS FOR THIS COLUMN TO MASTER:
    MSGTYPE = MSGTYPE + 1
    CALL PVMFINITSEND(PVMRAW,IBUFID)

C PACK ROW NUMBER:
    CALL PVMFPACK(INTEGER4,IROW+1,1,1,INFO)

C PACK COLUMN NUMBER:
    CALL PVMFPACK(INTEGER4,ICOL,1,1,INFO)

```

```

C   PACK NR COMPUTED ELEMENTS OF C:
      CALL PVMFPACK (REAL4, CIJ (1) , NR, 1, INFO)

C   SEND BUFFER TO MASTER PROGRAM:
      CALL PVMFSEND (MASTERNUM, MSGTYPE, INFO)
ENDDO

CALL PVMFEXIT (INFX)
END

```

Here, the  $J$  loop loops over the same space as the  $J$  loop in the master program. Within the loop, the slave receives columns of  $B$  into a single-dimensional array which it uses in the computation of an  $NR$ -row section of  $C$ . When these sections are computed, the slaves send them back, get another column of  $B$ , and the process is repeated.

---

### SPMD example

The example presented here uses the SPMD paradigm to perform a matrix addition. In it, the main program spawns a process for each processor, then the spawning process splits up the arrays into chunks and distribute them to the child processes. Both parent and child processes are controlled by the same program; the correct code is executed based on the presence or absence of a parent process, since only child processes have parents.

The main program follows:

```

PROGRAM SPMD
IMPLICIT NONE
INCLUDE "/USR/INCLUDE/FPVM3.H"

INTEGER NPROC, MAXROWS, MAXCOLS, MAXPROCS

PARAMETER (NPROC = 4, MAXROWS = 20, MAXCOLS = 20)
INTEGER A (MAXROWS, MAXCOLS), B (MAXROWS, MAXCOLS)
INTEGER C (MAXROWS, MAXCOLS)
COMMON / ARRAYBLK / A, B, C
CHARACTER*128 ERRMSG
CHARACTER*8 SPMD_EXE
INTEGER MYTID, NTASKS, TIDS (0:NPROC-1), I, J, TID
INTEGER STATUS, MYPROC, CHUNKID, BUFID, NUMBYTES

C ENROLL THE UNDER PVM:
CALL PVMFMYTID (MYTID)

```

```

C DETERMINE IF RUNNING PROCESS IS PARENT OR CHILD:
  CALL PVMFPARENT(TIDS(0)) ! ONLY SPAWNED PROCESSES HAVE PARENTS
  IF(TIDS(0) .EQ. PVMNOPARENT) THEN ! THIS PROCESS IS PARENT
    TIDS(0) = MYTID      ! SO PUT ITS ID INTO TIDS(0)
    MYPROC = 0

C   SPAWN A COPY FOR EACH PROCESSOR:
    SPMD_EXE = 'SPMD.EXE' ! NAME OF THIS PROGRAM
    CALL PVMFSPAWN(SPMD_EXE,PVMDEFAULT,'*',NPROC-1,TIDS(1),
1      NTASKS)

C   INITIALIZE ARRAYS A, B, AND C:
    CALL INIT_ARRAYS()

C   LOOP OVER SPAWNED TASKS:
    DO I = 2, NPROC
C     INITIALIZE SEND BUFFER:
        CALL PVMFINITSEND(PVMDEFAULT, STATUS)
C     PACK TIDS ARRAY ON SEND BUFFER:
        CALL PVMFPACK (INTEGER4, TIDS, NPROC, 1, STATUS)

C     NOTE: 5X20 SECTIONS OF A AND B ARE PACKED FOR EACH CHILD:
C     PACK A ARRAY SECTION ON SEND BUFFER:
        CALL PVMFPACK (INTEGER4, A(1, (I - 1) * 5 + 1),
1      MAXROWS * MAXCOLS / NPROC, 1, STATUS)

C     PACK B ARRAY SECTION ON SEND BUFFER:
        CALL PVMFPACK (INTEGER4, B(1, (I - 1) * 5 + 1),
1      MAXROWS * MAXCOLS / NPROC, 1, STATUS)

C     SET CHUNKID TO NUMBER OF CURRENT ITERATION (CHUNKID IS USED
C     BY SENDS AND RECEIVES TO DETERMINE WHERE A GIVEN ARRAY
C     CHUNK FITS INTO WHOLE ARRAY):
        CHUNKID = I

C     SEND MESSAGE BUFFER TO THE SPAWNED PROCESSES:
        CALL PVMFSEND (TIDS(I-1), CHUNKID, STATUS)
    END DO
  ELSE      ! THIS PROCESS IS A CHILD
C   NOTE NO LOOP REQ'D; EVERY CHILD RUNS THIS BRANCH IN PARALLEL
C   RECEIVE THE MESSAGE FROM THE PARENT PROCESS AND UNPACK IT:
    CALL PVMFRCV (TIDS(0), -1, BUFID)

```

```

C   GET INFORMATION ABOUT THE MESSAGE BUFFER RECEIVED
C   (EACH CHILD PROCESS RECEIVES A UNIQUE BUFFER)
      CALL PVMFBUFINFO (BUFID, NUMBYTES, CHUNKID, TID, STATUS)
      IF (STATUS .LT. 0) THEN
          ERRMSG = 'CHILD:PVMFBUFINFO FAILED'
          WRITE (6,*) 'STATUS = ', STATUS
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
      ELSE IF ((CHUNKID .LT. 2) .OR. (CHUNKID .GT. NPROC)) THEN
C   MAKE SURE CHUNKID IS VALID FOR CHILD PROCESS
          ERRMSG = 'CHILD:PVMFBUFINFO BOGUS MESSAGE TYPE'
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
      END IF

C   MESSAGE BUFFER IS OKAY; UNPACK:
C   UNPACK TIDS ARRAY:
      CALL PVMFUNPACK(INTEGER4, TIDS, NPROC, 1, STATUS)

C   UNPACK ARRAY A:
      CALL PVMFUNPACK(INTEGER4, A, (MAXROWS*MAXCOLS)/NPROC, 1, STATUS)

C   UNPACK ARRAY B:
      CALL PVMFUNPACK(INTEGER4, B, (MAXROWS*MAXCOLS)/NPROC, 1, STATUS)

C   FIND CURRENT PROCESS'S TID IN THE TID ARRAY:
      DO I = 1, NPROC-1
          IF (MYTID .EQ. TIDS(I)) THEN
              MYPROC = I ! ASSIGN PROCESS NUMBER FOR CURRENT PROCESS
              GOTO 100 ! AND EXIT LOOP
          END IF
      END DO

100  IF (MYPROC .LT. 1) THEN
          ERRMSG = 'CHILD:BOGUS PROCESS NUMBER'
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
      END IF

      ENDIF

C   ALL 'NPROC' PROCESSES ARE EQUAL NOW AND CAN BE ADDRESSED BY
C   TIDS(0) THRU TIDS(NPROC-1)

C   CALL COMPUTATION ROUTINE:
      CALL DOWORK(MYPROC, TIDS, CHUNKID, NPROC)

C   TERMINATE THIS PROGRAM.
      CALL PVMFEXIT (STATUS)
      CALL EXIT (0)
      END

```

Here, after spawning the specified number of child processes, the main program sends chunks of the A and B arrays to the appropriate children, and keeps a chunk of each for itself. The main program also contains code run by both parent and children which unpacks the array chunks and calls the DOWORK routine to perform the matrix addition.

The DOWORK subroutine is shown below.

```

SUBROUTINE DOWORK(MYPROC, TIDS, SENT_CHUNKID, NPROC)

C THIS ROUTINE IS CALLED BY EACH PROCESS (INCLUDING PARENT) TO DO
C MATRIX ADDITION ON THE PROCESS'S CHUNKS OF ARRAYS A, B AND C.
C AFTER CALCULATION, THE PARENT COLLECTS & OUTPUT RESULTS

    IMPLICIT NONE
    INCLUDE "/USR/INCLUDE/FPVM3.H"
    INTEGER NPROC, MAXROWS, MAXCOLS
    PARAMETER (MAXROWS = 20, MAXCOLS = 20)
    INTEGER A(MAXROWS, MAXCOLS), B(MAXROWS, MAXCOLS)
    INTEGER C(MAXROWS, MAXCOLS)
    COMMON / ARRAYBLK / A, B, C
    CHARACTER*128 ERRMSG
    INTEGER TIDS(0:NPROC-1), I, J, TID
    INTEGER STATUS, MYPROC, CHUNKID, BUFID, NUMBYTES, SENT_CHUNKID

C PERFORM MATRIX ADDITION USING CURRENT PROCESS'S SECTIONS:
    DO J = 1, MAXCOLS / NPROC
        DO I = 1, MAXROWS
            C(I, J) = A(I, J) + B(I, J)
        END DO
    END DO

C NOW CHILDREN SEND ARRAY SECTIONS TO PARENT, WHICH COLLECTS THEM:
    IF (MYPROC .EQ. 0) THEN ! CURRENT PROCESS IS PARENT
        DO I = 1, NPROC-1 ! LOOP OVER ALL PROCESS NUMBERS

C     MESSAGES BEING RETURNED FROM THE CHILD TASKS WILL HAVE
C     THE VALUE (SENT_CHUNKID + NPROC):
            CALL PVMFRECVC (-1, -1, BUFID)

C     GET INFORMATION ABOUT THE LAST MESSAGE BUFFER RECEIVED:
            CALL PVMFBUFINFO (BUFID, NUMBYTES, CHUNKID, TID, STATUS)
            IF (STATUS .LT. 0) THEN
                ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
                CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
            END IF
        END DO
    END IF

```

```

ELSE IF (NUMBYTES .NE. (MAXROWS*MAXCOLS*4)/NPROC) THEN
C CHECK THE LENGTH OF THE MESSAGE AND THE MESSAGE ID:
    ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
    CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
    ELSE IF ((CHUNKID.LT.6).OR.(CHUNKID.GT.2*NPROC)) THEN
C MAKE SURE CHUNKID IS VALID FOR CHILD PROCESS:
    ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
    CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
    END IF
C EXTRACT ARRAY C (RESULT OF MATRIX ADDITION) FROM BUFFER;
C NOTE CHUNKID AND NPROC ARE USED TO DETERMINE WHAT SECTION OF
C ARRAY IS BEING UNPACKED:
    CALL PVMFUNPACK(INTEGER4,C(1,(CHUNKID-NPROC-1)*5+1),
1          (MAXROWS*MAXCOLS)/NPROC,1,STATUS)
    END DO
ELSE ! CURRENT PROCESS IS CHILD
C SEND ARRAY C (RESULT ARRAY) SECTION BACK TO PARENT:
    CALL PVMFINITSEND(PVMDEFAULT,STATUS) !INITIALIZE SEND BUFFER
C PACK C ARRAY SECTION ON SEND BUFFER:
    CALL PVMFPACK(INTEGER4,C,(MAXROWS*MAXCOLS)/NPROC,1,STATUS)
C ENCODE ARRAY'S CHUNKID SO THAT PARENT'S UNPACK CAN DETERMINE
C WHICH CHUNK IT'S RECEIVING:
    CHUNKID = SENT_CHUNKID + NPROC
C SEND BUFFER TO PARENT PROCESS:
    CALL PVMFSEND (TIDS(0), CHUNKID, STATUS)
    END IF
C PARENT OUTPUTS RESULTS:
    IF (MYPROC .EQ. 0) CALL OUTPUT()
    RETURN
END

```

In DOWORK, the matrix addition is performed; then, in the ELSE clause of the following IF construct, the child processes send their chunks of the array C back to the parent. In the IF clause, the parent unpacks them and reassembles the entire C array. After this is done, the parent outputs the results.



This chapter discusses common optimization problems you may encounter when developing programs for SPP Series machines, and presents some possible solutions. Optimization can remove instructions, replace them, and change the order in which they execute. In some cases, improper optimizations can cause unexpected or incorrect results or code that slows down at higher optimization levels.

In some cases, user error can cause similar problems in code that contains syntactically correct constructs or directives that are used improperly.

If you encounter any of these problems, look for the following possible causes:

- Aliasing
- Invalid subscripts
- Floating-point imprecision (roundoff error)
- Misused directives and options
- Misused memory classes
- Compiler limitations

## Note

**Compilers perform optimizations assuming that the source code being compiled is valid. Optimizations done on source that violates certain ANSI standard rules can cause the compilers to generate incorrect code.**

---

## Aliasing in Fortran

As described in the "Inhibitors of localization" section of Chapter 3, "Compiler optimizations," an alias is an alternate name for some object. Fortran EQUIVALENCE statements, C pointers, and procedure calls in both languages can potentially cause aliasing problems. The examples presented in Chapter 3 concern aliasing problems that occur at optimization levels -O2 and above. However, code motion can also cause aliasing problems, at optimization levels -O1 and above.

Consider the following Fortran program:

```
PROGRAM ALIAS
INTEGER I
COMMON /DATA/I
I = 666
CALL CONFUSED(I)
END

SUBROUTINE CONFUSED(N1)
DO I = 1, 2
  N2 = 3 * (N1 + 1)
  CALL CALC(N2)
  WRITE(*,*) 'Iteration:', I, ', n1 = ', N1
  WRITE(*,*) 'Iteration:', I, ', n2 = ', N2
ENDDO
RETURN
END

SUBROUTINE CALC(N)
INTEGER K, N
COMMON /DATA/ K
K = N + 1
RETURN
END
```

In the subroutine CONFUSED, the compiler assumes that N1 is invariant. This is not true, as N1 is equivalent to I, which is in the DATA common block and therefore is manipulated by the CALC subroutine. The right side of the assignment to N2 appears to be loop invariant, so the compiler moves the assignment to N2 out of the loop at optimization levels -O1 and higher. When compiled at -O1 or above, the program produces incorrect answers.

The results of this program compiled and run at optimization levels `-O0` and `-O1` follow. Note that the answers are changed at optimization level `-O1`.

```
% fc -O0 alias.f -o O0.out
```

```
% O0.out
```

```
Iteration: 1, n1 = 2002
```

```
Iteration: 1, n2 = 2001
```

```
Iteration: 2, n1 = 6010
```

```
Iteration: 2, n2 = 6009
```

```
% fc -O1 alias.f -o O1.out
```

```
% O1.out
```

```
Iteration: 1, n1 = 2002
```

```
Iteration: 1, n2 = 2001
```

```
Iteration: 2, n1 = 2002
```

```
Iteration: 2, n2 = 2001
```

---

## Aliasing in C

Because they frequently use pointers, C programs are especially susceptible to aliasing problems. The C compiler has two different algorithms for detecting potential aliases: a “worst-case” algorithm, which is used in backward-compatible (`-pcc`) mode, and an ANSI-C aliasing algorithm, which is used in the ANSI-C compatibility modes (`-ext`, and `-std`).

---

### Worst-case algorithm

The worst-case aliasing algorithm views all pointer references as subscripts into a giant array that encompasses all of memory. This mythical array (known as `*MEM*` in the compiler’s optimization report) includes all global variables and local variables whose address is taken using the address (`&`) operator. This results in the following conditions:

- Every pointer reference is a potential alias for every other pointer reference.
- Every pointer reference is a potential alias for every global variable (and vice versa).
- Every pointer reference is a potential alias for every local variable that is used with `&`.

---

## ANSI algorithm

ANSI C provides stricter type-checking. This allows the C compiler to use a stricter algorithm that eliminates many potential aliases found by the worst-case algorithm. Instead of one giant \*MEM\* array, the ANSI-C algorithm uses a model with separate arrays for each base type (such as int, float, or double). Pointers and variables cannot alias with pointers or variables of a different base type.

The ANSI aliasing algorithm permits the compiler to parallelize the code shown below; the worst-case algorithm does not.

```
void alex1(a, ib, n)
float *a;
int *ib, n;
{
    int i;
    for (i=0; i<n; i++)
        a[i] = ib[i] + n;
}
```

Pointers a and ib point to different base types. The worst-case algorithm considers them to be aliased through \*MEM\*, but under ANSI C rules, no potential alias exists, and the loop is parallelized.

The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant. The non-ANSI-compliant code shown below allows two pointers of different base types to become aliased with one another. (The assignment to fptr makes the code non-ANSI C compliant.)

```
int array[100];
int *dummy;
float *fptr;

*dummy = array; /* illegal pointer/integer
                 combination. */
fptr = dummy; /* operands of = point to
              incompatible types. */
```

Compiling this code generates the warning messages shown in the comments. (Compiling with -d arg\_ptr\_ref=e converts these warnings into error messages.)

Because of the ANSI standard violation, this code does not compile safely under the ANSI aliasing algorithm.

In the following example, function caller passes two long int pointers to foo, which expects one pointer to long int and one pointer to short int. Because ia and ib have different base

types, the ANSI C aliasing algorithm assumes that no alias between *\*ia* and *\*ib* can exist.

At optimization level `-O3` in ANSI mode, the compiler compiles this code (with a warning) and parallelizes the loop even though aliasing and a recurrence do exist. In `-pcc` mode, the code will not compile.

```
void foo(long int *ia, short int *fb)
{
    int i;
    for (i=0; i<500; i++) {
        ia[i]=1;
        fb[i] = ia[i];
    }
}
caller()
{
    long int arra[600];
    foo(&arra[0], &arra[100]);
}
```

---

## Specifying aliasing modes

To specify an aliasing mode, use one of the following options on the `cc` command line:

- `-alias cautious`
- `-alias standard`
- `-alias worst`

In `cautious` mode, the compiler uses the ANSI C aliasing algorithm. If the compiler finds constructs that could cause hidden aliasing, it switches to the worst-case aliasing algorithm. If you do not specify an aliasing mode, the compiler uses `cautious` mode by default.

In `standard` mode, the compiler always uses the ANSI-C aliasing algorithm.

In `worst` mode, the compiler uses the worst-case aliasing algorithm.

These and other C aliasing options are further discussed in Appendix B, "Optimization options."

---

## Iteration and stop values

Aliasing a variable in an array subscript can make it unsafe for the compiler to parallelize a loop. Below are several situations that can prevent parallelization.

### Using potential aliases as addresses of variables

In the following example, the code passes `&j` to `getval`; `getval` can use that address in any number of ways, including possibly assigning it to `iptr`. (Even though `iptr` is not passed to `getval`, `getval` might still access it as a global variable or through another alias.) This situation makes `j` a potential alias for `*iptr`.

```
void subex(iptr, n, j)
int *iptr, n, j;
{
    n = getval(&j,n);

    for (j--; j<n; j++)
        iptr[j] += 1;
}
```

This potential alias means that `j` and `iptr[j]` might occupy the same memory space for some value of `j`. The assignment to `iptr[j]` on that iteration would also change the value of `j` itself. The possible alteration of `j` prevents the compiler from safely parallelizing the loop. In this case, the optimization report says that no induction variable could be found for the loop, and the compiler does not parallelize the loop.

Avoid taking the address of any variable that will be used as the iteration variable for a loop. To parallelize the loop in `subex`, use a temporary variable `i` as shown in the following example:

```
void subex(iptr, n, j)
int *iptr, n, j;
{
    int i;
    n = getval(&j,n);
    i=j;
    for (i--; i<n; i++)
        iptr[i] += 1;
}
```

### Using hidden aliases as pointers

In the next example, `ialex` takes the address of `j` and assigns it to `*ip`. Thus, `j` becomes an alias for `*ip` and, potentially, for `*iptr`. Assigned values to `iptr[j]` within the loop could alter

the value of `j`. As a result, the compiler cannot use `j` as an induction variable and, without an induction variable, it cannot count the iterations of the loop. When the compiler can't find the loop's iteration count, the compiler can't parallelize the loop.

```
int *ip
void ialex(iptr)
int *iptr;
{
    int j;
    *ip = &j;
    for (j=0; j<2048; j++)
        iptr[j] = 107;
}
```

To parallelize this loop, remove the line of code that takes the address of `j` or introduce a temporary variable.

### Using a pointer as a loop counter

Compiling the following function, the compiler finds that `*j` is not an induction variable (because an assignment to `iptr[*j]` could alter the value of `*j` within the loop.) The compiler does not parallelize the loop.

```
void ialex2(iptr, j, n)
int *iptr;
int *j, n;
{
    for (*j=0; *j<n; (*j)++)
        iptr[*j] = 107;
}
```

Again, this problem can be solved by introducing a temporary iteration variable.

### Aliasing stop variables

In the following code, the stop variable `n` becomes a possible alias for `*iptr` when `&n` is passed to `foo`. This means that `n` can be altered during the execution of the loop. As a result, the compiler cannot count the number of iterations and cannot parallelize the loop.

```

void salex(int *iptr, int n)
{
    int i;
    foo(&n);
    for (i=0; i<n; i++)
        iptr[i] += iptr[i];
    return;
}

```

To parallelize the affected loop, eliminate the call to `foo`, move the call below the loop (in which case flow-sensitive analysis takes care of the aliasing), or create a temporary variable as shown below:

```

void salex(int *iptr, int n)
{
    int i, tmp;
    foo(&n);
    tmp = n;
    for (i=0; i<tmp; i++)
        iptr[i] += iptr[i];
    return;
}

```

Because `tmp` is not aliased to `iptr`, the loop has a fixed stop value and the compiler parallelizes it.

---

## Global variables

Potential aliases involving global variables cause optimization problems in many programs. The compiler cannot tell whether another function causes a global variable to become aliased.

The following code uses a global variable, `n`, as a stop value. Because `n` may have its address taken and assigned to `ik` outside the scope of the function, `n` must be considered a potential alias for `*ik`. The value of `n`, therefore, can be altered on any iteration of the loop. The compiler cannot determine the stop value and cannot parallelize the loop.

```

int n, *ik;
void foo(int *ik)
{
    int i;

    for (i=0; i<n; i++)
        ik[i]=i;
}

```

Using a temporary local variable solves the problem.

```
int n;
void foo(int *ik)
{
    int i, stop = n;

    for (i=0; i<stop; ++i)
        ik[i]=i;
}
```

If `ik` is a global variable instead of a pointer, the problem does not occur. Global variables do not cause aliasing problems except when pointers are involved. The following code will parallelize.

```
int n, ik[1000];
void foo()
{
    int i;

    for (i=0; i<n; i++)
        ik[i] = i;
}
```

---

## Invalid subscripts

An array reference in which *any* subscript falls outside declared bounds for that dimension is called an invalid subscript. Invalid subscripts are a common cause of answers that vary between optimization levels and programs that abort and dump core.

---

## Floating-point imprecision

The compiler applies normal arithmetic rules to real numbers. It assumes that two arithmetically equivalent expressions produce the same numerical result.

Most real numbers cannot be represented exactly in digital computers. Instead, these numbers are rounded to a floating-point value that can be represented. When optimization changes the evaluation order of a floating-point expression, the results can change. Possible consequences of floating-point roundoff include program aborts, division by zero, address errors, and incorrect results.

In any parallel program, the execution order of the instructions will differ from the serial version of the same program. This can cause noticeable roundoff differences between the two versions. Running a parallel code under different machine configurations or conditions can also yield roundoff differences, because the execution order can differ under differing machine conditions, causing roundoff errors to propagate in different orders between

executions. Accumulator variables are especially susceptible to these problems. Consider the following Fortran example:

```
C$DIR GATE (ACCUM_LOCK)
      LK = ALLOC_GATE (ACCUM_LOCK)
      .
      .
      .
      LK = UNLOCK_GATE (ACCUM_LOCK)
C$DIR BEGIN_TASKS, TASK_PRIVATE (I)
      CALL COMPUTE (A)
C$DIR CRITICAL_SECTION (ACCUM_LOCK)
      ACCUM = ACCUM + A
C$DIR END_CRITICAL_SECTION
C$DIR NEXT_TASK
      DO I = 1, 10000
         B (I) = FUNC (I)
C$DIR CRITICAL_SECTION (ACCUM_LOCK)
      ACCUM = ACCUM + B (I)
C$DIR END_CRITICAL_SECTION
      .
      .
      .
      ENDDO

C$DIR NEXT_TASK
      DO I = 1, 10000
         X = C (I) + D (I)
      ENDDO
C$DIR CRITICAL_SECTION (ACCUM_LOCK)
      ACCUM = ACCUM / X
C$DIR END_CRITICAL_SECTION
C$DIR END_TASKS
```

Here, three parallel tasks are all manipulating the real variable ACCUM, using real variables which have themselves been manipulated. Each manipulation is subject to roundoff error, so the total roundoff error here might be substantial. When the program runs in serial, the tasks execute in their written order, and the roundoff errors accumulate in that order. However, if the tasks run in parallel, there is no guarantee as to what order the tasks will run in, meaning the roundoff error will accumulate in a different order than it does during the serial run. Depending on machine conditions, the tasks may run in different orders during different parallel runs also, potentially accumulating roundoff errors differently and yielding different answers.

An analogous C example follows:

```
static gate_t accum_lock;
lk = alloc_gate(&accum_lock);
.
.
.
lk = unlock_gate(&accum_lock);
#pragma _CNX begin_tasks, task_private(i)
compute(a);
#pragma _CNX critical_section(accum_lock)
accum = accum + a;
#pragma _CNX end_critical_section
#pragma _CNX next_task
for(i=0;i<10000;i++) {
    b[i] = func[i];
#pragma _CNX critical_section(accum_lock)
    accum = accum + b[i];
#pragma _CNX end_critical_section
.
.
.
}
#pragma _CNX next_task
for(i=0;i<10000;i++)
    x = c[i] + d[i];
#pragma _CNX critical_section(accum_lock)
accum = accum/x;
#pragma _CNX end_critical_section
#pragma _CNX end_tasks
```

Problems with floating-point precision can also occur when a program tests the value of a variable without allowing enough tolerance for roundoff errors. To solve the problem, adjust the tolerances to allow for greater roundoff errors or declare the variables to be of a higher precision (use the `double` type instead of `float` in C, or `REAL*8` rather than `REAL*4` in Fortran). It is always poor practice to test floating point numbers for exact equality.

---

## Disabling underflow traps

By default, PA-RISC processor hardware traps floating point underflow when a floating point result is *tiny*. A floating point number is considered tiny if its exponent field is zero but its mantissa is nonzero (for more information, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*). This practice is extremely costly in terms of execution time and seldom

provides any benefit. You can disable this behavior by passing the `+FPD` flag to the loader. This is done using the `-W` compiler option on either the `cc` or `fc` command line. The following example shows such an `fc` command line.

```
fc -Wl,+FPD prog.f
```

This command line compiles the program `prog.f` and instructs the loader to disable floating point underflow. For more information on the `-W` option, refer to the *Fortran User's Guide* or *C User's Guide*.

---

## Misused directives, pragmas, and options

Misused directives and pragmas are a common cause of wrong answers. For example, forcing parallelization of a loop containing a call is safe only if the called routine contains no dependencies.

Do not assume that it is always safe to parallelize a loop whose data is safe to localize. You can safely localize loop data in loops that do not contain a loop-carried dependency (LCD) of the form shown in the following Fortran loop:

```
DO I = 2, M
  DO J = 1, N
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

where one of `IADD` and `JADD` is negative and the other is positive. This is explained in detail in the "Inhibitors of localization" section of Chapter 3, "Compiler optimizations."

You cannot safely parallelize a loop that contains any kind of LCD. This is discussed further in the "Inhibitors of parallelization" section of Chapter 3, "Compiler optimizations."

The `MAIN` section of the Fortran program that follows initializes `A`, calls `CALC`, and outputs the new array values. In subroutine `CALC`, the indirect index used in `A(IN(I))` introduces a potential dependency that prevents the compiler from parallelizing `CALC`'s `I` loop.

```
PROGRAM MAIN
REAL A(1025)
INTEGER IN(1025)
COMMON /DATA/ A
DO I = 1, 1025
  IN(I) = I
ENDDO
```

```

CALL CALC(IN)
CALL OUTPUT(A)
END

SUBROUTINE CALC(IN)
INTEGER IN(1025)
REAL A(1025)
COMMON /DATA/ A
DO I = 1, 1025
    A(I) = A(IN(I))
ENDDO
RETURN
END

```

An analogous C example follows:

```

float arra[1025];

void calc(int in[])
{
    int i,j;

    for(i = 0; i <= 1024; i++)
        arra[i] = arra[in[i]];
}

main()
{
    int i,j,in[1025];

    for(i = 0; i <= 1024; i++)
        in[i] = i;
    calc(in);
    output(arr);
}

```

Because you know that  $IN(I) = I$ , you can use the `NO_LOOP_DEPENDENCE` directive, as shown below. This directive allows the compiler to ignore the apparent dependence and parallelize the loop at optimization level `-O3`.

```

SUBROUTINE CALC(IN)
INTEGER IN(1025)
REAL A(1025)
COMMON /DATA/ A
C$DIR NO_LOOP_DEPENDENCE(A)
DO I = 1, 1025
    A(I) = A(IN(I))
ENDDO
RETURN
END

```

In C:

```
void calc(int in[])
{
    int i,j;

#   pragma _CNX no_loop_dependence(arr)
    for(i = 0; i <= 1024; i++)
        arra[i] = arra[in[i]]
}
```

---

## Misused memory classes

While manually assigned memory classes can substantially boost performance when coupled with manual parallelization, assigning the wrong memory class to data can cause wrong answers and in some cases degrade performance. This section discusses some common misuses of memory classes.

---

### Improper dynamic allocations

Dynamically allocating `thread_private` memory from serial code can give unexpected results if the memory is later accessed from parallel code. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
   REAL*8 WRONGTP(:)
C$DIR THREAD_PRIVATE(WRONGTP)
   ALLOCATABLE WRONGTP
   .
   .
   .
C THE FOLLOWING ALLOCATE ONLY ALLOCATES
C WRONGTP(N) FOR THREAD 0:
   ALLOCATE(WRONGTP(N))
C$DIR LOOP_PARALLEL(THREADS, IVAR = I)
C$DIR LOOP_PRIVATE(J)
   DO I = 1, NUM_THREADS()
       DO J = 1, N
           WRONGTP(J) = ... ! ONLY EXISTS FOR
           .                 ! THREAD 0
           .
           .
       ENDDO
   ENDDO
```

Here, the array `WRONGTP` is allocated, but since the allocation takes place in serial code, which is run by thread 0, only thread 0

allocates the array. When other threads attempt to access the array in the  $J$  loop, it does not exist. To fix this, allocate the array inside the thread-parallel  $I$  loop, as shown in the example on page 158.

An analogous C example follows:

```

/* INCORRECT EXAMPLE FOLLOWS!!!                                     */
static thread_private double *wrongtp;
.
.
.
/* the following memory_class_malloc only allocates wrongtp
   for thread 0                                                    */
wrongtp=(double *)memory_class_malloc(sizeof(double)*n,
                                     THREAD_PRIVATE_MEM);
#pragma _CNX loop_parallel(threads, ivar = I)
#pragma _CNX loop_private(j)
for(i=0;i<num_threads();i++) {
  for(j=0;j<n;j++) {
    wrongtp[j] = ... /* only exists for thread 0 */
    .
    .
    .
  }
}

```

In general, memory of classes other than `thread_private` should be dynamically allocated in serial code. Allocating `node_private`, `near_shared`, `far_shared` and `block_shared` memory from within parallel code will create wasteful redundant copies. Consider the following *incorrect* Fortran example.

```

C INCORRECT EXAMPLE FOLLOWS!!!
  REAL*8 WRONGNP(:)
C$DIR NODE_PRIVATE(WRONGNP)
C$DIR FAR_SHARED_POINTER(WRONGNP)
  ALLOCATABLE(WRONGNP)
.
.
.
  N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES, IVAR = I)
  DO I = 1, N
    ALLOCATE(WRONGNP(M))
    .
    .
    .
  ENDDO

```

Recall from Chapter 5, “Memory classes,” that when a `node_private` array is allocated, a physical copy is created on each hypernode on which the program is running. Here, each loop iteration executes the `ALLOCATE` statement (or `memory_class_malloc` function in C), thus allocating  $N$  copies of the array. This is  $N-1$  times more copies than are actually needed. To further complicate things, `node_private` arrays manipulated in parallel code must be accessed by shared pointers, which is why the Fortran example includes a `far_shared_pointer` statement. In the code above, this pointer would be overwritten every time the `I` loop executed the `ALLOCATE` statement (or `memory_class_malloc` function in C), meaning that only the final copy allocated would be accessible. Since the hypernodes’ execution of the loop code is not perfectly synchronized, the actual memory accessed by `WRONGNP (I)` would vary depending on which hypernode was last to perform the allocation.

An analogous C example follows:

```

/* INCORRECT EXAMPLE FOLLOWS!!!                                     */
static far_shared double *wrongnp;
.
.
.
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar = i)
for(i=0;i<n;i++) {
    wrongnp = (double *)memory_class_malloc(sizeof(double)*m,
                                           NODE_PRIVATE_MEM);
.
.
.
}

```

While dynamically allocated `near_shared`, `far_shared` and `block_shared` arrays do not normally require special pointer types, they suffer from the same redundant-copy problem. Allocating any shared memory arrays from within parallel code will create as many copies of the data as there are hypernodes (or threads) executing the `ALLOCATE` (or `memory_class_malloc`) statement. As with the `node_private` example above, the actual memory accessed will depend on which hypernode most recently executed the `ALLOCATE` statement. After all hypernodes have executed the `ALLOCATE`, the memory allocated by all but the last will be lost. Such lost arrays are not only unusable, they cannot be deallocated.

To avoid such redundancy problems, follow the allocation examples discussed in Chapter 5, “Memory classes,” and only

allocate memory from within parallel constructs as described there.

---

## Incorrect array pointers

As mentioned in the previous section, sometimes it is necessary to access dynamically allocated arrays using pointers of different memory classes. For example, when accessing `node_private` arrays from `node-parallel` code, `far_shared` pointers must be used (refer to Chapter 5, “Memory classes”). Failing to do this will render the copies of the arrays on all but hypernode 0 inaccessible. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
   REAL*8 NONP(:)
C$DIR NODE_PRIVATE(NONP)
      ALLOCATABLE(NONP)
      .
      .
      .
      ALLOCATE(NONP(M))
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, N
C$DIR LOOP_PARALLEL(THREADS)
          DO J = 1, M
              NONP(J) = ...
          .
          .
          .
      ENDDO
ENDDO
```

While the `NONP` array is correctly allocated in serial code here, it is not explicitly given a shared pointer, so the arrays created will be accessed by the default `node_private` pointer. A physical copy of `NONP` will be created on every hypernode, but the `node_private` pointer by which these copies are accessed will only be initialized on hypernode 0, because it is the only hypernode executing the `ALLOCATE` statement (or `memory_class_malloc` in C). The contents of the (`node_private`) pointers on other hypernodes are uninitialized and therefore indeterminate. When, in the hypernode-parallel `J` loop, these other hypernodes attempt to access `NONP`, they will do so using the garbage contents of their uninitialized pointers, typically causing a runtime error.

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!! */
static node_private double *NONP;
.
.
.
n0np = (double *)memory_class_malloc(sizeof(double)*m,
                                     NODE_PRIVATE_MEM);
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar = i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar = j)
  for(j=0;j<m;j++) {
    n0np[j] = ...
    .
    .
    .
  }
}
```

Chapter 5, “Memory classes,” covers correct pointer/data combinations, and explains the situations in which non-default pointers should be used. To avoid uninitialized pointer problems such as the one described above, follow the recommendations of Chapter 5 carefully.

---

## Hidden dependencies

Improperly accessing a shared variable from parallel threads can create an unapparent dependency that can cause wrong answers.

Consider the following Fortran code:

```

PROGRAM HOLDER
  REAL HOLD
  C$DIR FAR_SHARED (HOLD)
  C$DIR TASK_PRIVATE (X, Y)
  C$DIR BEGIN_TASKS
    X = ...
    .
    .
    .
    CALL ADDHOLD (HOLD, X)
  C$DIR NEXT_TASK
    Y = ...
    .
    .
    .
    CALL ADDHOLD (HOLD, Y)
  C$DIR END_TASKS
  END

  SUBROUTINE ADDHOLD (HOLD, Z)
    REAL HOLD, Z
    HOLD = HOLD+Z
  END

```

Here, the `far_shared` variable `HOLD` is updated as a function of itself in the subroutine `ADDHOLD`, which is called from the potentially parallel tasks. If `HOLD` was updated within the tasks rather than in a subroutine, the dependency would be more obvious to the programmer, who may not have ready access to `ADDHOLD`'s source.

Isolating the assignment to `HOLD` inside a critical section would allow the tasks to safely parallelize, whether the assignment took place in a subroutine or inside the tasks themselves.

An analogous C example follows:

```

void addhold(float *hold, float z) {
    *hold = *hold + z;
}

```

```

main() {
    static far_shared float hold;
    static task_private float x,y;
    #pragma _CNX begin_tasks
    x = ...;
    .
    .
    .
    addhold(&hold,x);
    #pragma _CNX next_task
    y = ...;
    .
    .
    .
    addhold(&hold,y);
    #pragma _CNX end_tasks
}

```

Always use caution when parallelizing a call to a procedure that passes the same shared variable from every thread.

---

## Compiler limitations

Compiler limitations can produce faulty optimized code when the source code contains:

- Reductions
- Different possible evaluation orderings
- Iterations by zero
- Nondeterminism of parallel execution
- Replaceable loop test variables
- Trip counts greater than  $2^{31} - 1$  at optimization levels `-O2` and `-O3`
- Hidden ordered sections

---

## Reductions

Reductions are a special class of dependency that the compiler can parallelize. An apparent LCD can prevent the compiler from parallelizing a loop containing a reduction. The loop in the following Fortran example is not parallelized because of an apparent dependency between the references to `A(I)` on line 4 and the assignment to `A(JA(J))` on line 5. The compiler does not realize that the values of the elements of `JA` never coincide with the values of `I`, and so, assuming that they might, conservatively avoids parallelizing the loop.

```

DATA JA /11,12,13,14,15,16,17,18,19,20/
DO I = 1, 10
  DO J = I, 10
    A(I) = A(I) + B(J) * C(J)    !line 4
    A(JA(J)) = B(J) + C(J)      !line 5
  ENDDO
ENDDO

```

## Note

In this example as well as the examples that follow, the apparent dependence becomes real if any of the values of the elements of JA are equal the values iterated over by I.

A `no_loop_dependence` directive or pragma placed before the J loop tells the compiler that the indirect subscript does not cause a true dependence. Because reductions are a form of dependence, this directive also tells the compiler to ignore the reduction on A(I), which it would normally handle. Ignoring this reduction causes the compiler to generate incorrect code for the assignment on line 4; the apparent dependence on line 5 is properly handled because of the directive. The resulting code runs fast but produces incorrect answers.

In the following analogous C example, the apparent dependency is between the reference to a[i] on line 4 and a[ja[j]] on line 5:

```

ja[] = {11,12,13,14,15,16,17,18,19,20};
for (i=0; i<10; i++)
  for (j=0; j<10; j++) {
    a[i] += b[j] * c[j];    /* line 4 */
    a[ja[j]] = b[j] + c[j]; /* line 5 */
  }

```

To solve this problem, distribute the J loop, isolating the reduction from the other statements, as shown in the following Fortran example:

```

DATA JA/11,12,13,14,15,16,17,18,19,20/
DO I = 1, 10
  DO J = I, 10
    A(I) = A(I) + B(J) * C(J)
  ENDDO
ENDDO

C$DIR NO_LOOP_DEPENDENCE(A)
DO I = 1, 10
  DO J = I, 10
    A(JA(J)) = B(J) + C(J)
  ENDDO
ENDDO

```

And in C:

```
for (i=0; i<10; i++)
  for (j=i; j<10; j++)
    a[i] += b[j] * c[j];

#pragma _CNX no_loop_dependence(a)
for (i=0; i<10; i++)
  for (j=i; j<10; j++)
    a[ja[j]] = b[j] + c[j];
```

The apparent dependency is removed, and both loops can be optimized.

This problem occurs only if the reduction and the apparent dependency involve the same variable or array element. If the reduction and the apparent dependency involve different variables or array elements, as in the following Fortran example, both reduction and dependency are handled correctly without your intervention.

```
DATA JD /6, 7, 8, 9, 10/
DO I = 1, 5
C$DIR NO_LOOP_DEPENDENCE(D)
  DO J = I, 5
    A(I) = A(I) + B(J) * C(J)
    D(JD(J)) = D(I) + B(J) + C(J)
  ENDDO
ENDDO
```

In C:

```
jd[] = {6,7,8,9,10};
for (i=0; i<5; i++)
# pragma _CNX no_loop_dependence(d)
  for (j=0; j<5; j++){
    a[i] += b[j] * c[j];
    d[jd[j]] = d[i] + b[j] + c[j];
  }
```

---

## Evaluation order

Assumptions the compiler makes about reordering code can sometimes cause answers to change at higher optimization levels. If this happens, use parentheses to force a specific order of evaluation.

## Incrementing by zero

If the compiler parallelizes a loop that increments a variable by zero on each trip, the loop can produce incorrect answers or cause the program to abort. This error can occur when a variable used as an incrementation value is accidentally set to zero. If the compiler detects that the variable has been set to zero, the compiler does not parallelize the loop. If the compiler cannot detect the assignment, however, the previously described symptoms occur. The following Fortran example shows three loops that increment by zero.

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)

J = 1
DO I = 1, N
    B(I) = A(J)
    A(J) = C(I)
    J = J + IZR
ENDDO

DO I = 1, N, IZR    ! INCREMENT VALUE OF 0 IS
                  ! NON-STANDARD

    A(I) = B(I)
ENDDO

DO I = 1, N
    J = J + IZR
    B(I) = A(J)
    A(J) = C(I)
ENDDO
```

Because IZR is an argument passed to SUB1, the compiler does not detect that IZR has been set to zero. All three loops parallelize at -O3, but because of the zero increments, their runtime behavior cannot be reliably predicted. All three loops compile at -O1, but the second loop, which specifies the step as part of the DO statement (or as part of the for statement in C), will cause a runtime error. Runtime behavior of the other two loops cannot be predicted at -O1.

The analogous C code follows:

```
float a[100],b[100],c[100];

void sub1(int izr)
{
    int i,j = 1;

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

main()
{
    sub1(0);
}
```

---

## Nondeterminism of parallel execution

In a parallel program, threads do not execute in a predictable or determined order. If you force the compiler to parallelize a loop when a dependency exists, the results are unpredictable and can vary from one execution to the next.

Consider the following Fortran example:

```
DO I = 1, N-1
    A(I) = A(I+1) * B(I)
    .
    .
    .
ENDDO
```

The compiler will not parallelize this code as written because of the dependency on  $A(I)$ . This dependency requires that the original value of  $A(I+1)$  is available for the computation of  $A(I)$ . If this code was parallelized, some values of  $A$  would be assigned by some processors before they were used by others, resulting in incorrect assignments. Because the results depend on the order in

which statements execute, the errors are nondeterministic. The loop must therefore execute in iteration order to ensure that all values of A are computed correctly.

The analogous C code follows:

```
for (i=0; i<n-1; i++) {  
    a[i] = a[i+1] * b[i];  
    .  
    .  
    .  
}
```

Loops containing dependencies can sometimes be manually parallelized using the `LOOP_PARALLEL (ORDERED)` directive as described in Chapter 6, “Advanced shared memory programming.” Otherwise, unless you are sure that no loop carried dependency exists, it is safest to let the compiler choose which loops to parallelize.

---

## Large trip counts at -O1 and above

When a loop is optimized at level `-O1` or above, its trip count must occupy no more than a signed 32-bit storage location. The largest positive value that can fit in this space is  $2^{31} - 1$  (2,147,483,647). If the compiler can determine that the trip count is larger than this at compile time, it will issue a warning. Loops with trip counts that cannot be determined at compile time but that exceed  $2^{31} - 1$  at runtime will yield wrong answers.

This limitation only applies at optimization levels `-O1` and above.

Loops with trip counts that overflow 32 bits can be optimized by manually strip mining the loop.

---

## Hidden ordered sections

While it is legal and sometimes useful to place ordered sections in separate routines from their parent ordered loops, this practice can cause runtime deadlock in some situations.

Consider the following Fortran example:

```
PROGRAM SEPMAIN
REAL A(100)
.
.
.
C$DIR BEGIN_TASKS
CALL SUB1(A)
.
.
.
C$DIR NEXT_TASK
CALL SUBN
.
.
.
C$DIR END_TASKS
.
.
.
END

SUBROUTINE SUB1(A)
REAL A(100)
C$DIR GATE(LOCK)
LK = ALLOC_GATE(LOCK)
C$DIR LOOP_PARALLEL(ORDERED)
DO I = 2, 100
    CALL SUB2(LOCK,A,I)
ENDDO
LK = FREE_GATE(LOCK)
END

SUBROUTINE SUB2(LOCK,A,I)
C$DIR GATE(LOCK)
REAL A(100)
INTEGER I
C$DIR ORDERED_SECTION(LOCK)
A(I) = A(I-1)
C$DIR END_ORDERED_SECTION
END
```

Here, the tasks in the main program go thread parallel by default, so when the loop in SUB1 is reached it cannot go parallel. However, because parallelism exists in the main program, the ordered section in SUB2 expects that it will be executed by all parallel threads. Only thread 0 is executing SUB1, since only the first ordered task calls it; thread 0 therefore runs the DO loop in SUB1 and passes through the ordered section in SUB2. After this,

the ordered section will wait for thread 1 to enter before allowing thread 0 back in on the next iteration of the `I` loop. Thread 1 never calls `SUB1`, so it never has the opportunity to enter the ordered section. This causes the program to deadlock in the ordered section.

The analogous C code follows:

```
void sub1(float *a) {
    static gate_t lock;
    int lk;
    lk = alloc_gate(&lock);
#pragma _CNX loop_parallel(ordered, ivar=1)
    for(i=1;i<100;i++)
        sub2(lock,a,i);
    lk = free_gate(&lock);
}

void sub2(gate_t lock, float *a, int i) {
#pragma ordered_section(lock)
    a[i] = a[i-1];
#pragma end_ordered_section
}

main() {
    float a[100];
    .
    .
    .
#pragma _CNX begin_tasks
    sub1(a[]);
    .
    .
    .
#pragma _CNX next_task
    subn();
    .
    .
    .
#pragma _CNX end_tasks
    .
    .
    .
}
```

If you encounter this kind of problem, try moving the ordered section into the same routine as its parent loop, or, if possible, add another dimension of parallelism so that the loop running the ordered section is running in parallel.



---

# Potentially unsafe optimizations

# 9

This chapter describes optimizations that can potentially generate incorrect results. By default, the SPP Series Fortran and C compilers avoid performing these optimizations. You can enable potentially unsafe optimizations by specifying the `-uo` compiler option.

The `-uo` option enables the compiler to perform these optimizations:

- Simple strength reductions
- Code motion
- Elimination of type conversions

---

## Simple strength reduction

Chapter 3, “Compiler optimizations,” describes how the compiler replaces slow operations with faster ones on the assumption that arithmetically equivalent expressions always yield the same results. However, reducing an expression such as  $X/C$  to  $(1/C) * X$  can be unsafe because it can increase roundoff error.

When you use the `-uo` option, the compiler replaces division operations with multiplication. If a possibility of overflow exists, however, the compiler does not perform this optimization.

---

## Code motion

The compiler normally moves an invariant expression out of a loop only if the expression is located on a path to all loop exits. When you use `-uo`, the compiler can move an invariant expression out of a loop if the expression does not lie on a path to all loop exits.

In the following example, the invariant expression  $A=B/X$  is relocated only when the program is compiled with the `-uo` option:

```
DO I = 1, 100
  IF (X .NE. 0) THEN
    A= B/X
    AR(I) = A*C
  ELSE
    AR(I) = D*C
  ENDIF
ENDDO
```

---

## Conversion elimination

Type conversions are costly in terms of machine cycles, and they can inhibit optimization. When you use the `-uo` option, the compiler eliminates costly type conversions by creating REAL induction variables that it then increments concurrently with the loop's INTEGER induction variables. Consider the following Fortran loop:

```
REAL A(100000)
.
.
.
DO I = 1, 100000
  A(I) = I
ENDDO
```

And its C equivalent:

```
float a[100000];

int foo()
{
  int I;

  for(I=0; I<100000; I++)
    a[I] = I;
}
```

Here, in absence of the `-uo` option, `I` must be converted to floating point on every iteration of the loop. With the `-uo` option the compiler avoids this costly operation by copying `I` into a REAL (or float) induction variable before entering the loop, then incrementing this variable by `1.0` on every iteration of the loop. At optimization level `-O3`, the compiler can then parallelize the following optimized Fortran loop:

```
REAL_I = 1.0
I = 1
10  A(I) = REAL_I
    REAL_I = REAL_I + 1.0
    I = I + 1
    IF (I .LE. 100000) GOTO 10
```

**In C:**

```
float a[100000];

int foo()
{
    int I;
    float REAL_I;

    for(I=0, REAL_I=0.0; I<100000; I++, REAL_I++)
        a[I] = REAL_I;
}
```

This optimization is considered potentially unsafe because the internal representation of real numbers is inexact, and this can lead to a significant accumulated error when `REAL_I` is incremented over the course of the loop.



---

# Compiler directives and pragmas

# A

This appendix presents an alphabetical list of all the Fortran directives and C pragmas that are supported by the Convex SPP Series compilers.

This appendix is intended to provide only a brief overview of the available directives and pragmas. More specific information and examples can be found elsewhere in this guide. The Fortran directives not supported as C pragmas are expressed as storage class extensions (`thread_private`, etc.) or as typedefs (`gate_t`, `barrier_t`, etc.), and are described in Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared memory programming.”

The form of an SPP Series Fortran compiler directive is:

```
C$DIR [CSERIES|SPP] directive-specification
```

The form of an SPP Series C pragma is:

```
#pragma _CNX [CSERIES|SPP] directive-specification
```

Where *directive-specification* is one of the directives/pragmas described in this chapter. `CSERIES` or `SPP` can optionally be included to indicate the target machine for the directive; this is useful if you are writing code that may be compiled on either C Series or Exemplar architectures, and wish to include directives or pragmas for both without having the compiler issue warnings for those that do not apply to the current machine.

Directive names are presented here in lower case; they may be specified in either case in both languages, but “#pragma” must always appear in lower case in C. In the sections that follow, *namelist* represents a comma delimited list of names. These names can be variables, arrays, or common blocks. In the case of a common block, its name must be enclosed within slashes. The occurrence of a lower-case *n* is used to indicate a compile-time

integer constant expression; values not determinable at compile time cannot be used. Occurrences of *gate\_var* are for variables that have been, or are being, defined as gates. Any parameters that appear within square brackets ( [ and ] ) are optional.

---

### **align\_cti ( *namelist* )**

Aligns the variables and arrays listed in *namelist* on CTIcache boundaries. This allows for more efficient data reuse.

---

### **barrier( *namelist* )**

This Fortran directive is used to denote a list of variables, as given in *namelist*, that will be used as the synchronization variables for the barrier routines. This does not imply any synchronization in itself, it is simply defining the barrier variables. Note that in C, *barrier* is a typedef, rather than a pragma. For more information, refer to Chapter 6, "Advanced shared memory programming."

---

### **begin\_tasks [ ( *attribute\_list* ) ]**

This directive or pragma defines the beginning of a section (or sections; see *next\_task*) of code that will be executed as an independent, parallel task. Each task is executed by a separate thread. *begin\_tasks* must have an accompanying *end\_tasks* in the same program unit.

The optional *attribute\_list* can be any of the following legal combinations:

- (max\_threads=*m*)
- (ordered)
- (nodes)
- (threads)
- (ordered, nodes)
- (ordered, threads)
- (ordered, max\_threads=*m*)
- (nodes, max\_threads=*m*)
- (threads, max\_threads=*m*)
- (ordered, nodes, max\_threads=*m*)
- (ordered, threads, max\_threads=*m*)

Refer to Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming,” for a complete discussion of parallel tasking.

---

### **block\_loop [ (block\_factor=*n*) ]**

This directive or pragma is used to indicate a specific loop to block, and optionally, the block factor that will be used in the compiler’s internal computation of loop nest based data reuse. In absence of the `block_factor` argument, this directive is useful for indicating which loop in a nest the compiler should block. Refer to Chapter 3, “Compiler optimizations,” for more information on blocking.

---

### **block\_shared (allocatable\_array\_namelist)**

This Fortran directive is used to declare arrays as being of type block-shared. Block-shared arrays are sized to be an integral multiple of the page size. The pages of the array are distributed in same-size blocks across the hypernodes on which the process is executing in the subcomplex. If the user specified size is not an integral multiple of `page_size × num_nodes ( )` then the compiler will automatically round it up to meet this criterion. Refer to Chapter 5, “Memory classes,” for more information.

---

### **critical\_section [ (gate\_var) ]**

This defines the beginning of a code block in which only one thread may be executing at a time. The end of the code block must be indicated by an `end_critical_section` directive or pragma, which must appear in the same flow of control within the same program unit. The optional `gate_var` can be used to differentiate between parallel tasks. Refer to Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming,” for more information.

---

### **dynsel [ (trip\_count=*n*) ]**

Enables workload-based dynamic selection for the immediately following loop. `trip_count` represents either the `thread_trip_count` or `node_trip_count` attribute. If `thread_trip_count = n` is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. If `node_trip_count = n` is

specified, the serial version of the loop is run if the iteration count is less than  $n$ ; otherwise, the hypernode-parallel version is run.  $n$  must be a compile-time constant.

---

### **end\_critical\_section**

This directive or pragma is used to define the end of the critical section that was begun with the `critical_section` directive or `pragma.critical_section` and `end_critical_section` must appear as a pair. Refer to Chapter 4, "Basic shared memory programming," and Chapter 6, "Advanced shared memory programming," for more information.

---

### **end\_ordered\_section**

This directive or pragma is used to define the end of the ordered section that was begun with the `ordered_section` directive or `pragma.ordered_section` and `end_ordered_section` must appear as a pair. Refer to Chapter 6, "Advanced shared memory programming," for more information on ordered sections.

---

### **end\_tasks**

This is used to terminate the specification of parallel tasks indicated by `begin_tasks` and `next_task`. It must appear at the end of the last section of parallel code defined by these directives or pragmas. All of these must appear in the same program unit. Refer to Chapter 4, "Basic shared memory programming," and Chapter 6, "Advanced shared memory programming," for more information.

---

### **far\_shared(namelist)**

This Fortran directive causes the compiler to place the data objects in *namelist* (i.e., variables, arrays, or common blocks) into `far_shared` memory. `far_shared` memory is the most general form that is distributed on a page basis across the memories of all hypernodes in a subcomplex. The `far_shared` data objects of a process are addressable by all threads of that process. Refer to Chapter 5, "Memory classes," for more information on memory classes.

---

## **far\_shared\_pointer**(*alloc\_var\_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `far_shared` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

---

## **gate**(*namelist*)

This Fortran directive defines a gate variable that will be subsequently used in a critical section, ordered section, or passed as an argument to the synchronization intrinsics. Note that in C, `gate` is a typedef, rather than a pragma. Refer to Chapter 6, “Advanced shared memory programming,” for more information.

---

## **loop\_parallel** [ (*attribute\_list*) ]

This is an explicit instruction to the compiler to parallelize the immediately following loop. The loop iterations will be run in an indeterminate order unless the optional `ordered` attribute appears. The user is responsible for any required data privatization and loop synchronization, as described in Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming.” The optional *attribute\_list* can be any of the following combinations:

- (`chunk_size=n`)
- (`max_threads=m`)
- (`ordered`)
- (`nodes`)
- (`threads`)
- (`ordered, nodes`)
- (`ordered, threads`)
- (`nodes, chunk_size=n`)
- (`threads, chunk_size=n`)
- (`chunk_size=n, max_threads=m`)
- (`ordered, max_threads=m`)
- (`nodes, max_threads=m`)
- (`threads, max_threads=m`)

- (ordered, nodes, max\_threads=*m*)
- (ordered, threads, max\_threads=*m*)
- (nodes, chunk\_size=*n*, max\_threads=*m*)
- (threads, chunk\_size=*n*, max\_threads=*m*)
- (ivar = *indvar*)—Required for all loops in C and for DO WHILE and hand rolled loops in Fortran; optional for Fortran DO loops; can be specified with any other attribute.

Attributes may be listed in any order. Any attribute combinations other than those listed above will be flagged with a fatal error by the compilers. Refer to Chapter 6, “Advanced shared memory programming,” for more information.

---

### **loop\_private** (*namelist*)

This is used to declare a list of variables and/or arrays private to the immediately following loop. To be loop private, the variables and/or arrays must be assigned before they are used on any iteration of the immediately following loop. These private data items are distinct from the shared items of the same name that exist outside the loop. No values may be carried into the loop by loop\_private variables. Values assigned to loop\_private variables on the final iteration may be saved into the shared variables of the same name if the save\_last directive or pragma also appears on this loop. If save\_last is not used, then the value of any shared variable declared to be loop\_private is undefined at loop termination. Refer to Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming,” for more information.

---

### **near\_shared** (*namelist*)

When applied to static variables at compile-time, this Fortran directive will cause all pages of the data objects in *namelist* to be mapped to physical pages on hypernode 0. If applied to allocatable arrays, then the pages of such arrays will be mapped to physical pages on the hypernode of the allocating thread. near\_shared data can be addressed by any thread of a process on any hypernode in the subcomplex but it is “closer” (in terms of access latency) to the threads on the hypernode that allocates the data. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

---

## **near\_shared\_pointer(*alloc\_var\_name*)**

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `near_shared` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information on memory classes.

---

## **next\_task**

This starts a block of code following a `begin_tasks` block that will be executed as a parallel task. The end of the code block is marked by another `next_task` or by an `end_tasks` directive or `pragma`.

This directive must appear within a `begin_tasks` and `end_tasks` pair. There is no limit on the number of `next_task` directives that can appear. Refer to Chapter 4, "Basic shared memory programming," and Chapter 6, "Advanced shared memory programming," for more information.

---

## **no\_block\_loop**

Informs the compiler to perform no blocking on the immediately following loop. Generally the compiler will attempt to automatically block a nested loop in order to achieve optimal cache data reuse between loop iterations. Refer to Chapter 3, "Compiler optimizations," for more information on loop blocking.

---

## **no\_distribute**

Disables loop distribution for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information on loop distribution.

---

## **no\_dynsel**

Disables workload-based dynamic selection for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information on dynamic selection.

---

## **no\_fuse**

Disables loop fusion for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information on loop fusion.

---

## **no\_loop\_dependence (namelist)**

Informs the compiler that the arrays in *namelist* do not have any dependencies for iterations of the immediately following loop. Use `no_loop_dependence` for arrays only; use `loop_private` to indicate dependence-free scalar variables.

This will cause the compiler to ignore any dependencies that it perceives to exist. This can enhance the compiler’s ability to optimize the loop, including the possibility of parallelization.

Refer to Chapter 3, “Compiler optimizations,” and Chapter 8, “Limits of optimization,” for more information.

---

## **no\_parallel**

Prevents the compiler from generating parallel code for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **no\_peel**

Prevents the compiler from peeling the immediately following loop. Loop peeling involves removing tests that appear for the first and/or last iterations of a loop to outside the loop body. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **no\_promote\_test**

Prevents the compiler from performing test promotion on the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **no\_side\_effects(*funclist*)**

Informs the compiler that the functions appearing in *funclist* have no side effects wherever they appear lexically following the directive. Side effects include modifying a function argument, modifying a Fortran COMMON variable, performing I/O, or calling another routine that does any of the above. The compiler can sometimes eliminate calls to procedures that have no side effects.

---

## **no\_unroll**

Disables loop unrolling. Refer to the “Loop unrolling” section of Chapter 3, “Compiler optimizations,” for more information.

---

## **no\_unroll\_and\_jam**

Disables unrolling and jamming of loops. Refer to the “Loop unroll and jam” section of Chapter 3, “Compiler optimizations,” for more information.

---

## **node\_private(*namelist*)**

This Fortran directive causes the variables and arrays specified in *namelist* to be replicated in the physical memory of each hypernode on which the process is executing. Thus, while each data object has a single image in virtual memory, it maps to a different physical location on each hypernode. Process threads within a hypernode all share access to the copy on their hypernode and cannot access the copies on other hypernodes. Refer to Chapter 5, “Memory classes,” for more information.

---

## **node\_private\_pointer(*alloc\_var\_name*)**

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in

`node_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information.

---

### **opt\_level (level)**

This C pragma forces the optimization level to be the lower of the level supplied on the command line and the level supplied in *level* for all routines following the pragma, which must appear at file scope. Consider the following example:

```
#pragma _CNX opt_level(-O3)
func() {
    .
    .
    .
}
#pragma _CNX opt_level(-O1)
```

If the file containing this code was compiled using the `-O2` command line argument, the function `func` would be compiled at optimization level `-O2` and functions following the second `opt_level` pragma would be compiled at optimization level `-O1`. `opt_level` has no effect if it appears inside a function. No Fortran directive version of `opt_level` is available.

---

### **ordered\_section (gate\_var)**

An ordered section is the same as a critical section with the additional restriction that the threads must pass through the ordered section in iteration order. Ordered sections must appear within the control flow of a `loop_parallel (ordered)`. Refer to Chapter 6, "Advanced shared memory programming," for more information.

---

### **peel\_all**

This has the same effect as the `peel` directive or pragma except that it permits the compiler to replicate code without bound. Normally peeling will only replicate a certain amount of code. This can slow the compile phase and might even cause internal compiler tables to overflow. Refer to Chapter 3, "Compiler optimizations," for more information.

---

## **peel**

Allows the compiler to peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound. Refer to Chapter 3 for more information on peeling. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **prefer\_fuse**

Specifies fusion for particular loops. A `prefer_fuse` directive or pragma must immediately precede the loop for which you want to enable fusion. If fusion is enabled for the program, `prefer_fuse` has no affect. If fusion is disabled for the program, `prefer_fuse` must be specified on two or more neighboring loops, and will override the `-nfl` option. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **prefer\_parallel**

Instructs the compiler to parallelize the following loop but only if it is safe to do so. A loop is safe to parallelize if it has an iteration count determinable at runtime before loop invocation, and contains no LCDs, procedure calls, or I/O operations. Refer to Chapter 4, “Basic shared memory programming,” for more information. Refer to Chapter 4, “Basic shared memory programming,” for more information.

---

## **promote\_test\_all**

Instructs the compiler to promote tests out of the immediately following loop. The amount of code replication is not constrained in any way. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

## **promote\_test**

Instructs the compiler to promote tests out of the immediately following loop, expanding code beyond the default conservative limit but not without bound. Refer to Chapter 3, “Compiler optimizations,” for more information.

---

### **row\_wise (array\_namelist)**

In Fortran, the normal convention of storing arrays in column major order (left-most index varies fastest) can be switched for the arrays specified in *array\_namelist*. Specifying *row\_wise* for an array causes it to be stored in row major order which is consistent with the C language storage ordering.

---

### **save\_last**

This specifies that all variables named in an associated *loop\_private (namelist)* or *task\_private (namelist)* must have their last value saved into the "shared" variable of the same name, at loop or task termination. If *save\_last* is not specified then the values in any privatized variables or arrays are indeterminate at loop termination. Refer to Chapter 6, "Advanced shared memory programming," for more information.

---

### **scalar**

This directive or pragma prevents the compiler from performing reordering transformations on the following loop; the compiler will not parallelize or data-localize a loop on which this directive appears.

---

### **sync\_routine (funclist)**

This indicates to the compiler that the functions listed in *funclist* are user-defined synchronization functions, so that the compiler will not attempt to move code across these function calls. Use *sync\_routine* anytime you hide a call to a compiler synchronization function inside another procedure call, or anytime you use CPSlib functions for synchronization.

*sync\_routine* is only effective for the listed procedures which lexically follow it in the procedure in which it appears.

---

### **task\_private (namelist)**

This will privatize the variables and arrays specified in *namelist* for each task specified in the immediately following *begin\_tasks/end\_tasks* block. The privatized variables and arrays will not carry their values beyond the *end\_tasks*

directive or pragma. Refer to Chapter 4, “Basic shared memory programming,” and Chapter 6, “Advanced shared memory programming,” for more information.

---

### **thread\_private (*namelist*)**

This Fortran directive will cause the variables and arrays specified in *namelist* to be treated (by software convention) as being `thread_private`. `thread_private` data objects map to unique `node_private` addresses for each thread of a process. Refer to Chapter 5, “Memory classes,” for more information.

---

### **thread\_private\_pointer (*alloc\_var\_name*)**

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `thread_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, “Memory classes,” for more information.

---

### **unroll [ (*unroll\_factor*=*n*) ]**

This causes the immediately following loop to have its body replicated (*n* times if `unroll_factor` is specified) and unrolled in order to reduce loop overhead.

This only takes place if

- the loop is a scalar loop
- there is no internal branching
- the specified loop is innermost.

Complete unrolling occurs if the loop count is less than 5. Otherwise only partial unrolling occurs. When used on a loop nest, this directive or pragma must be placed on the loop that ends up being innermost after compilation, or it will have no effect.

Refer to the “Loop unrolling” section of Chapter 3, “Compiler optimizations,” for more information.

---

### **unroll\_and\_jam [ (*unroll\_factor*=*n*) ]**

This causes one or more non-innermost loops in the immediately following nest to be partially unrolled (to a depth of *n* if `unroll_factor` is specified), then fuses the resulting loops back

together. It must be placed on a loop that ends up being non-innermost after any compiler-initiated interchanges. Refer to the “Loop unroll and jam” section of Chapter 3, “Compiler optimizations,” for more information.

This appendix lists the optimization options available for use with Convex SPP Series Fortran and C compilers and briefly describes each option. A complete list of all Fortran compiler options is available in Chapter 1, "Compiling programs," of the *Convex Fortran User's Guide*. A complete list of all C compiler options is available in Chapter 2, "Compiler fundamentals," of the *Convex C User's Guide*.

---

## Optimization level and related options

The options listed in this section specify the level of optimization allowed.

-no

Machine instruction level scalar optimization. This option is the default.

-O0

Basic block level scalar optimization.

-O1

-O0 optimizations plus program unit level scalar optimization and global register allocation.

-O2

-O1 plus data localization.

-O3

-O2 plus parallelization.

-or *table*

Specifies the contents of the optimization report. Values of *table* and the optimization reports they produce are shown in Table 7.

**Table 7** Optimization report contents

Table value	Report contents
all	Loop table, privatization table and array table
loop	Loop table only (default)
array	Array table only
private	Loop table and privatization table
none	No report

For more information about the optimization report, refer to Appendix C, "Optimization report."

`-noautopar`

Disables parallelization for any loop not specifically parallelized by a `loop_parallel` or `prefer_parallel` directive or pragma. Loops preceded by these directives are parallelized; all other loops are not parallelized. This is only effective when specified with the `-O3` option.

---

## Cross compilation options

The options listed in this section allow a program to be optimized for a machine configuration that differs from the configuration of the machine the program is being compiled on.

`-tm target`

Specifies the target machine architecture for which compilation is to be performed. *target* takes the value `spp1000` to specify SPP1000 Series machines (the default), or `spp1200` for SPP1200 Series machines.

`-cache n`

Instructs the compiler to assume a per-processor direct-mapped cache size of *n* kbytes. The compiler uses this information when determining a loop's blocking factor. Cache size defaults to the size of the cache on which the program is being compiled. This option may be useful when cross-compiling for future SPP Series machines which may have different cache sizes.

---

## Loop replication options

The options listed in this section control loop replication optimizations.

`-mr1`

Raises the limit on replicated loops. This may significantly increase compile time. If you aren't concerned with long compile times, you can use this option when the compiler issues advisories which indicate that loop replication is being limited for certain optimizations. Optimizations that replicate loops include `IF-DO` and `if-for` optimizations, dynamic selection, loop unrolling, `unroll` and `jam`, and loop blocking. This option is available only at optimization level `-O2` or `-O3`.

`-ur`

Causes the compiler to automatically find unrollable loops and unroll them. Loops with iteration counts determinable at compile time to be less than 5 are unrolled completely. Those with indeterminate iteration counts, or determinate counts of 5 or more, are partially unrolled. Only innermost loops can be unrolled. This option is the default, and is available only at optimization levels `-O2` or `-O3`.

`-urn n`

Enables loop unrolling with an unroll factor of *n*; *n* is the number of times to replicate the body of the loop.

`-nur`

Disables loop unrolling.

`-uj`

Enables the unroll and jam optimization. This optimization unrolls one or more non-innermost loops in a nest, then jams the resulting loops back together, greatly improving register usage. This option is the default, and is available only at optimization levels `-O2` and `-O3`. For more information, refer to the "Loop unroll and jam" section of Chapter 3, "Compiler optimizations."

`-ujn n`

Enables unroll and jam with an unroll factor of *n*.

`-nuj`

Disables unroll and jam.

`-ds`

Enables workload-based dynamic selection. This optimization causes the compiler to generate parallel and serial versions of parallelizable loops whose iteration counts are unknown at compile time. At runtime, the loop's

workload is compared to parallelization overhead, and the parallel version is run only if it is profitable to do so.

`-nds`

Disables dynamic selection.

---

## Loop blocking options

The options listed in this section control the loop blocking optimization.

`-blockloop n`

Instructs the compiler to use a block factor of *n* and to automatically select which loops to block. In a loop nest containing *m* loops, *m*-1 loops will be blocked. If the `-blockloop` option is not used, the compiler will attempt to choose an optimal blocking factor based on the cache size and the amount of data being manipulated by the loop. Loop blocking is provided at optimization levels `-O2` and `-O3` unless the `-noblock` option is specified.

`-noblock`

Disables loop blocking for the sources being compiled. Because loop blocking occurs at optimization levels `-O2` and `-O3`, the `-noblock` option is effective only at these levels.

---

## IF-DO and if-for optimization options

The options listed in this section control the degree of loop peeling and test promotion allowed.

`-nopeel`

Disallows loop boundary value peeling, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to the `-peel` and `-peelall` options described in this section.

`-noptst`

Disallows test promotion, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to `-ptst` and `-ptstall` below.

`-peel`

Removes the first and/or last iterations of a loop when doing so removes conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates to `.TRUE.` or `.FALSE.` for the first and/or last iteration. By default, the compiler peels boundary values and expands code up to a predetermined conservative limit. With the `-peel` option, this limit is increased and code expansion may become

significant. `-peel` must be used with the `-O2` or `-O3` optimization options.

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can greatly lengthen compiler time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options.

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticeable increase in compile time. `-ptst` must be used with the `-O2` or `-O3` optimization options.

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, this can significantly increase compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-ptstall` must be used with the `-O2` or `-O3` optimization options.

---

## Register use options

The options listed in this section control the extent to which registers are exploited.

`-nga`

Disable global register allocation for arguments passed by reference. This is useful for programs that violate the ANSI standard by passing a constant as a procedure argument when the procedure may potentially write to the dummy argument. Refer to the "Global register allocation" section of Chapter 3, "Compiler optimizations," for more information.

`-ngs`

Disable global register allocation for shared memory variables which are visible to multiple threads. This option may help if a variable shared among parallel threads is causing wrong answers. Refer to the "Global register allocation" section of Chapter 3, "Compiler optimizations," for more information.

---

## Data alignment options

The options listed below allow you to control data alignment.

### `-align cseries`

Causes the Fortran compiler to store `COMMON` blocks using "tight" packing (the ANSI standard). Tight `COMMON` block packing is the default on Convex C Series machines.

Rather than padding `COMMON` block data items to their natural boundaries (the default on SPP Series machines), this method stores them in memory contiguously, so partial-word items may cause other data items to align on unnatural boundaries. For instance, a `REAL*8` item may align on a 2- or 4-byte boundary instead of an 8-byte boundary. Note that this natural boundary alignment is in addition to the CTIcache boundary alignment discussed in the "Data alignment" section of Chapter 2, "Architecture overview."

If you specify tight packing, you must take other measures to prevent runtime errors as described in the *Release Notice* for the compiler you are using.

### `-align cti`

Aligns all arrays on CTIcache boundaries, increasing the efficiency of data reuse.

### `-align spp`

The opposite of `-align cseries` and is the default on SPP Series machines.

For more information, refer to the "COMMON block packing" section of Chapter 5, "Specification statements," of the *Convex Fortran Language Reference*.

---

## C aliasing options

Potential aliases occur more frequently in C than Fortran because of C's typically heavy use of pointers. This section covers the options supported by the Convex C compiler to help you deal with aliasing problems.

### `-alias array_args`

Causes the compiler to assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but can allow greater optimization to occur. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to the elements of the formal parameter (for example, `formalParameter[10] = x[10]`).

The following source code illustrates a situation in which the `-alias array_args` option is useful:

```
void vaf(char []);
char global = 'A';

int main()
{
    char b[10];

    vaf(b);
    return(1);
}

void vaf(char array[])
{
    int i;

    for(i=0; i<10; i++)
        array[i] += global;
}
```

The loop in the `vaf` function can be parallelized when the `-alias array_args` option is specified because the compiler can assume that the address of `global` is not the address of an element of array.

`-alias standard`

Performs aliasing based on assumptions permitted in ANSI C; pointers of one object type can only reference objects of that same type. For example, a pointer to an object of type `float` cannot reference an object of type `int`. Such assumptions permit additional optimizations. If this option is specified when such conditions do not exist, the resulting program may not function correctly.

`-alias cautious`

Tries to compile with `-alias standard`, but if inappropriate constructs are found, compiles with `-alias worst` instead. This is the default in the ANSI C-compatible modes.

`-alias worst`

Performs pointer aliasing based on the assumption that pointers can modify objects of any type. This is the default in the backward-compatible (`-pcc`) mode.

The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`.

`-alias [no_global | global]`

When `-alias no_global` is specified, the compiler assumes that globals (external variables) are not accessed via pointers. The default is `global`.

Compiling the following code with `-alias no_global` would be an *incorrect* use of the option because the value of the global is accessed through the pointer `gp` as well as the variable `glob`.

```
int glob; /* wrong answers result if
           compiled with -alias
           no_global!!!          */
int *gp;

bar()
{
    gp = &glob;
}

main()
{
    int tmp;

    bar();
    glob = 1;
    *gp = 2;
    tmp = glob;
    printf("%d\n", tmp);
}
```

The program may print 1 when compiled with `-alias no_global`.

`-alias [no_addr | addr]`

`-alias no_addr`

Indicates that address operands do not introduce aliases. This is useful when your program passes data by reference and you are sure that no aliases are produced. The default is `-alias addr`.

`-alias ptr_args`

Causes the compiler to assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but allows greater optimization to occur. This option cannot be used if the formal parameter itself is

assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to variables identified by the formal parameter (for example, `formalParameter[10] = x[10]`).

**-alias restrict\_args**

Causes the compiler to treat the code as though a `restrict` qualifier was applied to each pointer parameter. The `restrict` qualifier tells the compiler that a pointer provides exclusive access to the data object at a given memory location. When you use the `restrict` qualifier, you must access the object pointed to by a `restrict` pointer only through that pointer. If you reference the object by some other means, the compiler may incorrectly optimize code.

---

## Other optimization options

This section lists optimization options that cannot be otherwise categorized.

**-il**

Instructs the Fortran compiler to prepare an intermediate language (`.fil`) file for a subprogram that is to be used for inline substitution. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization levels are ignored. Refer to the *Convex Fortran User's Guide*, Chapter 1, "Compiling programs."

**-is directory**

Instructs the Fortran compiler to attempt inline substitution of each subprogram for which there exists a `.fil` (intermediate-language file) file in the specified *directory*. This option must be repeated for each directory containing `.fil` files to be used for inline substitution. Refer to the *Convex Fortran User's Guide*, Chapter 1, "Compiling programs."

**-nore**

SPP Series compilers compile all procedures for reentrancy by default (refer to the `-re` option). Fortran procedures can be compiled using the `-nore` compiler option, which causes nonreentrant compilation. Procedures compiled in this manner cannot be parallelized or specified in a `NO_SIDE_EFFECTS` directive or pragma.

Many Fortran compilers, including the Convex C Series Fortran compiler, save local procedure variables by default. If your program is expecting this behavior, default reentrant compilation may cause wrong answers. Compiling the offending procedures with `-nore` should correct the problem.

-nsr

Disable scalar replacement.

-re

Causes reentrant compilation, which is the default. Reentrant procedures store all local variables on the stack; no values can be carried from one invocation of the procedure to the next. This allows parallel procedure calls.

-sr

Enable scalar replacement. This is the default at optimization level -O2 and above. Refer to the "Scalar replacement" section of Chapter 3, "Compiler optimizations."

-uo

Performs potentially unsafe optimizations, for example, moving the evaluation of common subexpressions or invariant code from within conditionally executed code. This moved code may be executed unconditionally. Refer to Chapter 9, "Potentially unsafe optimizations."



This appendix provides a complete description of the optimization report produced by the Convex SPP Series C and Fortran compilers. When you compile a program with the `-O2` or `-O3` option, the compiler generates an optimization report for each program unit. The `-or table` option determines the report's contents based on the value of *table*, as shown in Table 8.

**Table 8** Optimization report contents

Table value	Report contents
all	Loop table, privatization table and array table
loop	Loop table only (default)
array	Array table only
private	Loop table and privatization table
none	No report

## Loop table

The loop table lists the optimizations that were performed on each loop and, if appropriate, the reasons why a possible optimization was not performed. Loop nests are reported in the order in which they are encountered, and separated by a blank line. A description of each column of the loop table follows.

Line Num.

Specifies the source line of the beginning of the loop, or of the loop from which it was derived. If the line number has two parts separated by a hyphen, the second part is the distributed number (due to loop distribution).

Id Num.

Specifies a unique ID number for every loop. This ID number can then be referenced by other parts of the report. Both loops appearing in the original program source and loops created

by the compiler are given loop ID numbers; loops created by the compiler are also enumerated in the `New Loops` column as described further on. No distinction between compiler-generated loops and loops that existed in the original source is made in the `Id Num` column; loops are assigned unique, sequential numbers as they are encountered.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop. If the variable is compiler-generated, its name is listed as `*VAR*`; if there is no iteration variable, it is listed as `*NONE*`. If the iteration variable has two parts separated by a colon, the second part is the inline substitution instance of that variable. If it consists of a truncated variable name followed by a colon and a number, the number is a reference to the variable name footnote table which appears after the loop table, analysis table, and test table in the report.

`Reordering Transformation`

Indicates which reordering transformations were performed. Reordering transformations are performed on loops and loop nests, and typically involve reordering and/or duplicating sections of code to facilitate more efficient execution. This column has one of the values shown in Table 9.

Table 9 Reordering transformations reported in opt. report

Value	Explanation
Serial	No reordering transformation was performed.
PARALLEL	The loop runs in parallel mode.
Interchange	Loop interchange was performed. The new loop order may be indicated under the Optimizing/Special Transformation column, as shown in .
Dist	Loop distribution was performed.
DynSel	Dynamic selection was performed. The numbers in the New Loops column correspond to the loops created; for reentrant parallel loops, these generally include a PARALLEL and a Serial version.
Peel	Loop peeling was performed. In addition to cases in which loops are peeled to remove invariant boundary-iteration assignments, this may appear for automatically parallelized loops when it is necessary to peel the last iteration in order to assign automatically privatized variables their last-iteration value.
Promote	Test promotion was performed.
*	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, peeling, interchange, promotion).

#### New Loops

Specifies the loop ID number(s) for loops created by the compiler. These ID numbers are listed in the `Id Num.` column and can be referenced in other parts of the report; however, the loops they represent were not present in the original source code. In the case of loop fusion, the number in this column indicates the loop into which the current loop was fused.

#### Optimizing/Special Transformation

Indicates which, if any, optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with

simpler operations. A special transformation allows the compiler to optimize code under special circumstances. When appropriate, this column has one of the values shown in Table 10.

**Table 10** Optimizing/special transformations in opt. report

Value	Explanation
Unroll	The loop was completely or partially unrolled.
Reduction	Compiler recognized a reduction in the loop.
Pattern	The compiler recognized a special pattern in the loop.
Removed	The compiler removed the loop.
StripMine	The loop was strip mined.
Blocked	The loop was blocked.
Fused	The loop was fused.
Reentrant	Dynamic selection was performed for reentrancy.
Work	Workload-based dynamic selection was performed.
( <i>oldorder</i> ) -> ( <i>neworder</i> )	This may appear when Interchange is reported under Reordering Transformation. <i>oldorder</i> indicates the order of loops in the original nest; <i>neworder</i> indicates the new order. <i>oldorder</i> and <i>neworder</i> consist of user iteration variables presented in outermost to innermost order; if user variables are not available, line numbers are used. If the multiple nests begin on the same line, as with compiler-generated loops associated with Fortran 90 array assignments, this information is not reported.

### Supplemental tables

The tables described in this section are included if necessary to provide information supplemental to the loop table.

## Analysis table

If necessary, an analysis table is included in the optimization report to further elaborate on optimizations reported in the loop table. A description of each column of the analysis table follows.

Line Num.

Specifies the source line of the beginning of the loop.

Id Num.

References the ID number assigned to the loop in the loop table.

Iter. Var.

Specifies the name of the iteration variable controlling the loop, \*VAR\*, or \*NONE\*, as described in the "Loop table" section of this appendix.

Analysis

Indicates why a transformation or optimization was not performed, or additional information on what was done.

## Test table

If any test promotion or removal optimizations were performed, a test table is included in the optimization report. A description of each column in the test table follows.

Line Num.

Specifies the source line number of the beginning of the IF test.

Col. Num.

Specifies the source column number of the beginning of the IF test.

Test Transformation

Specifies the transformation performed: either TEST PROMOTED or TEST REMOVED.

Analysis

Presents a further explanation of the transformation performed, including the source line number of the original loop from which the test was transformed, and (if applicable), in parentheses, the loop ID number of the compiler-generated loop from which the test was transformed. This ID will be missing if the loop was removed.

## Privatization table

This table reports any user variables contained in a parallelized loop that are privatized by the compiler. Because the privatization table refers to loops, the loop table is automatically provided with it. A description of each column in the privatization table follows.

Line Num.

Specifies the source line of the beginning of the loop.

Id Num.

References the ID number assigned to the loop in the loop table.

Iter. Var.

Specifies the name of the iteration variable controlling the loop, \*VAR\*, or \*NONE\*, as described in the "Loop table" section of this appendix.

Priv. Var.

Specifies the name of the privatized user variable. Compiler-generated variables that are privatized are not reported here.

Privatization Information for Parallel Loops

Provides more detail on the privatization performed. For example, this column may indicate that a variable was automatically privatized by the compiler and that its last assignment was peeled from the loop so that its final value could be saved for later use.

## Variable name footnote table

Variable names that are too long to fit in the *Iter. Var.* columns of the other tables are truncated and followed by a colon and a footnote number. These footnotes are explained in the variable name footnote table. The headings in the variable name footnote table are explained below.

Footnoted *Iter. Var.*

Specifies the truncated variable name and its footnote number.

User Variable Name

Specifies the actual name of the variable as given by the user in the source code.

## Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed. The array table contains the following information.

Line Num.

Specifies the source line on which the reference occurs.

Var. Name

Specifies the name of the array being referenced.

Optimization

Describes the optimizations, if any, performed on the array in question.

Dependencies

If an array or memory dependency prevented optimization, this column shows the names of variables in the recurrence, in the form *name@linenumber*. If the reference could be to any memory location, it is in the form *\*MEM\*@linenumber*. If the reference is to a subprogram call, it is in the form *\*CALL\*@linenumber*.

## Examples

The following Fortran examples enumerate the contents of the optimization report. In discussing the examples, loops are referred to by their ID numbers.

While only Fortran examples are given, analogous C code would produce similar optimization reports.

### Example 1

Consider the following loop (line numbers are provided for reference):

```

1  SUBROUTINE EXAMPLE1 (A, B, C, N)
2  REAL A(N,N), B(N,N), C(N)

3  DO ILOOPINDEX = 1, N
4    C(ILOOPINDEX) = ILOOPINDEX
5    DO JLOOPINDEX = 1, N
6      A(ILOOPINDEX, JLOOPINDEX) = B(JLOOPINDEX, ILOOPINDEX)
7      + C(ILOOPINDEX)
8    ENDDO
9  ENDDO
10 END
```

Figure 24 shows the optimization report generated by compiling the program EXAMPLE1 at optimization level -O3.

```

% fc -O3 -or all example1.f

      Optimization for EXAMPLE1

Line   Id   Iter.   Reordering   New   Optimizing / Special
Num.   Num.  Var.    Transformation  Loops Transformation
-----
   3     1  ILOOPI:1 *Dist                (2-3)  No Strip
  3-1    2  ILOOPI:1 *DynSel             (4-5)  Work Reentrant
   3     4  ILOOPI:1 PARALLEL                Unroll

   3     5  ILOOPI:1 Serial                Unroll

  3-2    3  ILOOPI:1 *DynSel             (6-7)  Work Reentrant
   3     6  ILOOPI:1 *Interchange        (8)    (ILOOPI:1 JLOOPI:2
                                           JLOOPI:2) -> (JLOOPI:2
                                           ILOOPI:1 JLOOPI:2)
                                           Blocked Unroll

   5     8  JLOOPI:2 PARALLEL                Unroll
   3     9  ILOOPI:1 Serial

   3     7  ILOOPI:1 *Interchange        (10)   (ILOOPI:1 JLOOPI:2
                                           JLOOPI:2) -> (JLOOPI:2
                                           ILOOPI:1 JLOOPI:2)
                                           Blocked Unroll

   5    10  JLOOPI:2 Serial                Unroll
   3    11  ILOOPI:1 Serial

Line   Id   Iter.   Analysis
Num.   Num.  Var.

-----
   5     8  JLOOPI:2Loop blocked by 224 iterations
   5    10  JLOOPI:2Loop blocked by 224 iterations
   3     4  ILOOPI:1Replicated: partially unrolled with factor of 8
   3     5  ILOOPI:1Replicated: partially unrolled with factor of 8
   5     8  JLOOPI:2Replicated: partially unrolled with factor of 5
   5    10  JLOOPI:2Replicated: partially unrolled with factor of 5

Line   Id   Iter.   Priv.   Privatization Information
Num.   Num.  Var.    Var.    for Parallel Loops
-----
   3     4  ILOOPI:1 ILOOPI:1 Loop induction variable privatized
   3     4  ILOOPI:1 ILOOPI:1 Loop induction variable privatized
   5     8  JLOOPI:2 JLOOPI:2 Loop induction variable privatized
   5     8  JLOOPI:2 ILOOPI:1 Reference privatized

```

Figure 24 Optimization report for Example 1

Footnoted	User Variable
Iter. Var.	Name

ILOOPI:1	ILOOPINDEX
JLOOPI:2	JLOOPINDEX

Array References for Procedure EXAMPLE

Line Num.	Var. Name.	Optimi- zation	Dependencies
6	C	SR Hoist	
6	C	SR Hoist	
6	C	SR Hoist	
6	C	SR Hoist	

Figure 24 Optimization report for Example 1 (continued)

### Loop table

EXAMPLE1's loop table reports the following:

- Loop number 1 is the loop that appears on line 3 of the source. It iterates over the variable ILOOPINDEX. It was distributed and two new loops, numbers 2 and 3, were created. Loop 1 was not stripmined.

Note that according to the variable name footnote table, ILOOPINDEX is abbreviated as ILOOPI:1; similarly, JLOOPINDEX is abbreviated as JLOOPI:2. These names are used throughout the report to refer to these iteration variables.

- 3-1, the line number for loop number 2, tells us that it came from the loop at line 3 of the source, and that it is in the first distributed part of that distribution (this is indicated by the -1). Two loops, 4 and 5, are created to facilitate dynamic selection of loop 2. This dynamic selection is workload-based and is required for reentrancy. Loop 4 is the parallel version of loop 2, and loop 5 is the serial version. Both of these loops are unrolled.
- The line number 3-2 for loop 3 tells us that it is the second part of the distribution of the loop appearing on line 3 of the source (loop 2). Like loop 2, both workload-based and reentrant dynamic selection is performed on loop 3, resulting in the creation of loops 6 and 7.
- Loop 6 is interchanged; the Optimizing/Special Transformation column shows the loop nest order before and after interchange. There are two JLOOPINDEX loops

before interchange as a result of the blocking that is reported for loop 8 (optimizations are not always reported in the order in which they are applied). The innermost `JLOOPINDEX` loop is interchanged to outside the `ILOOPINDEX` loop, and this creates loop 8.

- Loop 8 is the final parallel version of the now-outer `JLOOPINDEX` loop. It is blocked and unrolled. Loop 8 contains within it loop 9, which is the `ILOOPINDEX` loop after interchange.
- Loop 7 is interchanged just as loop 6 is, but instead of creating a parallel loop, it creates a serial loop, loop 10.
- Loop 10 is the final serial version of the now-outer `JLOOPINDEX` loop, and includes within it loop 11, which is the `ILOOPINDEX` loop after interchange.

### Analysis table

EXAMPLE1's analysis table reports the following:

- Loops 8 and 10 are blocked by 224 iterations.
- Loops 4 and 5 are partially unrolled with an unroll factor of 8.
- Loops 8 and 10 are partially unrolled with an unroll factor of 5.

### Privatization table

EXAMPLE1's privatization table reports the following:

- The induction variable `ILOOPINDEX` is privatized in loop 4. This is reported twice because the unrolling of loop 4 creates two loops, but only one of them is shown in the optimization report.
- The induction variable `JLOOPINDEX` is privatized in loop 8.
- The induction variable `ILOOPINDEX` is also privatized in loop 8 because the `ILOOPINDEX` loop is nested within the `JLOOPINDEX` loop in loop 8.

### Array reference table

The array reference table indicates that the reference to the C array on line 6 was subject to scalar replacement (SR) and the resulting register load was hoisted out of the innermost loop. There are four entries because this optimization occurred for 4 loops (some of which were compiler-generated) during compilation.

---

## Example 2

The following Fortran code provides an example of other transformations the compiler performs. (Line numbers are listed for reference.)

```
1  SUBROUTINE EXAMPLE2 (A, N, ZERO, NEGATE, SUM)
2  REAL A(N), SUM
3  LOGICAL ZERO, NEGATE
4
5  SUM = 0.0
6  DO I = 1, N
7      SUM = SUM + A(I)
8      IF (ZERO) THEN
9          A(I) = 0
10     ELSE IF (NEGATE) THEN
11         A(I) = -A(I)
12     ENDIF
13     IF (I .EQ. 1 .OR. I .EQ. N) THEN
14         A(I) = -1
15     ENDIF
16 ENDDO
17 END
```

Figure 25 shows the optimization report generated by compiling the subroutine EXAMPLE2 above for parallelization.

```
% fc -03 -or all example2.f
```

Optimization for EXAMPLE2

Line Num.	Id Num.	Iter. Var.	Reordering Transformation	New Loops	Optimizing / Special Transformation
6	1	I	*Promote	(2-3)	
6	2	I	*Promote	(4-5)	
6	4	I	*Peel	(6)	
6	6	I	*DynSel	(7-8)	Work Reentrant
6	7	I	PARALLEL		Unroll
6	8	I	Serial		Unroll
6	5	I	*Peel	(9)	
6	9	I	*DynSel	(10-11)	Work Reentrant
6	10	I	PARALLEL		Unroll
6	11	I	Serial		Unroll
6	3	I	*Peel	(12)	
6	12	I	*DynSel	(13-14)	Work Reentrant
6	13	I	PARALLEL		Unroll
6	14	I	Serial		Unroll

Line Num.	Id Num.	Iter. Var.	Analysis
6	7	I	Replicated: partially unrolled with factor of 8
6	8	I	Replicated: partially unrolled with factor of 8
6	10	I	Replicated: partially unrolled with factor of 8
6	11	I	Replicated: partially unrolled with factor of 8
6	13	I	Replicated: partially unrolled with factor of 8
6	14	I	Replicated: partially unrolled with factor of 8

Line Num.	Col. Num.	Test Transformation	Analysis
8	14	TEST PROMOTED	Test promoted out of loop on 6 (7)
10	19	TEST PROMOTED	Test promoted out of loop on 6 (7)
13	16	TEST REMOVED	Peeled first iteration of loop on 6 (13)
13	16	TEST REMOVED	Peeled first iteration of loop on 6 (7)
13	16	TEST REMOVED	Peeled first iteration of loop on 6 (10)
13	30	TEST REMOVED	Peeled last iteration of loop on 6 (13)
13	30	TEST REMOVED	Peeled last iteration of loop on 6 (7)
13	30	TEST REMOVED	Peeled last iteration of loop on 6 (10)

Figure 25 Optimization report for Example 2

Line Num.	Id Num.	Iter. Var.	Priv. Var.	Privatization Information for Parallel Loops
6	7	I	SUM	Scalar reduction privatized, value saved
6	7	I	I	Loop induction variable privatized
6	7	I	SUM	Scalar reduction privatized, value saved
6	7	I	I	Loop induction variable privatized
6	10	I	SUM	Scalar reduction privatized, value saved
6	10	I	I	Loop induction variable privatized
6	10	I	SUM	Scalar reduction privatized, value saved
6	10	I	I	Loop induction variable privatized
6	13	I	SUM	Scalar reduction privatized, value saved
6	13	I	I	Loop induction variable privatized
6	13	I	SUM	Scalar reduction privatized, value saved
6	13	I	I	Loop induction variable privatized

#### Array References for Procedure EXAMPLE2

Line Num.	Var. Name.	Optimization	Dependencies
7	A	SR Hoist	
7	A	SR Hoist	
7	A	SR Hoist	
.	.	.	.
.	.	.	.
.	.	.	.

Figure 25 Optimization report for Example 2 (continued)

### Loop table

EXAMPLE2's loop table reports the following:

- There is only one loop in this example, and it appears at line 6 in the source. Loops 2 and 3, which are noted under New Loops on loop 1's line, are the result of promoting tests from the original loop; loops 4 and 5 are the result of promoting tests from loop 2.
- Loop 6 is created when loop 4 is peeled. Two loops, 7 and 8, are then created to facilitate workload-based and reentrant dynamic selection of loop 6. Loop 7 is the parallel version of loop 6, and is unrolled. Loop 8 is the serial version of loop 6; it is also unrolled.
- Loops 3 and 5 were both peeled. Peeling loop 5 produces loop 9; peeling loop 3 produces loop 12. Workload-based and reentrant dynamic selection is then applied to both resulting

loops. This replicates loop 9 into loops 10 and 11. Loop 12 is replicated into loops 13 and 14.

- Loop 10 is the parallel version of loop 9, and loop 11 is the serial version. Loop 13 is the parallel version of loop 12, and loop 14 is the serial version.

After all the transformations are completed, six loops (ID numbers 7, 8, 10, 11, 13, and 14) remain in the program. These remaining loops can be easily spotted under the `Reordering Transformation` column, as they are the loops that are not marked with the "\*" transformation indicator. Loops marked with this symbol no longer exist because they are replaced by the new loops indicated in the `New Loops` column.

### Analysis table

The analysis table of this report gives details of the test promotions and peelings that are mentioned in the loop table. It reports that each of the loops remaining after all transformations are performed is unrolled with an unroll factor of 8.

### Test table

EXAMPLE2's test table gives details of the test promotions and removals that are indicated in the loop table by the keyword `Peel`. The test table reports the following:

- The tests on lines 8 and 10 are promoted out of loop 7.
- The tests on line 13 are removed (the `.OR.` operator implies two tests) by peeling. This optimization is performed for every remaining parallel loop after all the transformations are completed (loops 7, 10, and 13), and it is reported separately for each loop.

### Privatization table

For each loop, the table contains an entry indicating that the induction variable `I` was privatized, and a second entry indicating the privatization of a scalar reduction (involving the variable `SUM`). This indicates that `SUM` was accumulated separately in a private copy in each parallel thread, and these copies were automatically added together after the threads joined. Privatization is only performed in parallel loops, so this is only reported for the parallel loops that remain after all transformations are complete (loops 7, 10, and 13).

### Array table

The references to array `A` on lines 7, 9 and 11 are subject to scalar replacement and hoisting, and this is reported for every loop

which includes these lines, including compiler-generated copies (the report has been shortened for brevity).



---

## Introduction

The Convex Compiler Parallel Support Library (CPSlib) is a library of thread management and synchronization routines which can be used to control parallelism on Exemplar systems. Most programs can fully exploit their parallelism via higher-level devices such as automatic parallelization, compiler directives, and ConvexPVM; CPSlib is provided for those few cases in which a lower-level interface is required. Using CPSlib requires you to manually control all aspects of parallelism, synchronization, and data partitioning.

This appendix includes a discussion of the forms of parallelism available via CPSlib, instructions for accessing CPSlib, a brief description of each of the routines included in CPSlib, and examples of common programming constructs as implemented using CPSlib routines. For further information, refer to the section 3 man pages for the routine in question, or to the `cps(3)` man page for an overview.

CPSlib supports two forms of parallelism: symmetric and asymmetric.

---

## Symmetric parallelism

In symmetric parallelism, several threads execute the same instruction stream. Symmetric parallelism is typically used by the compilers to parallelize a loop; the description of parallelism given in Chapter 3, "Compiler optimizations," is a description of symmetric parallelism.

Symmetric parallel threads are spawned using `cps_ppcall()` or `cps_ppcalln()`, which, along with all CPSlib routines, are described in detail further on. These functions automatically spawn a given number of threads, which call a specified routine in

parallel. All parallel work must occur in the called routine. When the routine returns, `cps_ppcall()` or `cps_ppcalln()` automatically executes a join and the program proceeds in serial.

Figure 26 shows a `cps_ppcall()` that spawns two threads, each of which in turn spawns four threads. The arrows represent a thread's instruction flow; the numbers labeling the arrows indicate the spawn thread ids. Kernel and spawn thread IDs are also discussed in the "Thread ID assignments" section of Chapter 6, "Advanced shared memory programming."

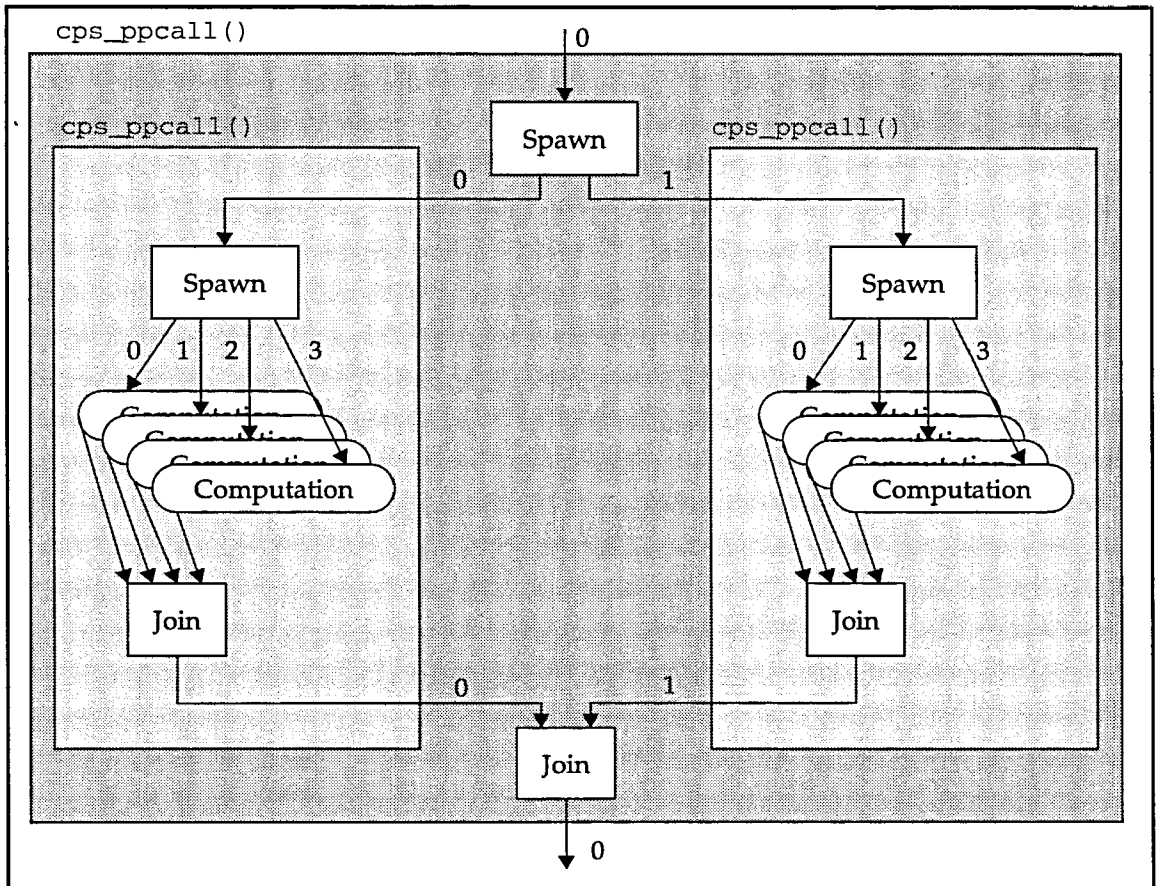


Figure 26 Symmetric parallelism

Shaded boxes represent operations hidden from the user by `cps_ppcall()`. As shown in the left branch of the first spawn, when a `cps_ppcall()` or `cps_ppcalln()` is processed, the parent thread is allocated to the computation as spawn thread ID 0; its kernel thread ID, which is uninteresting to the user, is unchanged. Additional peer threads in both spawned threads are spawned and assigned spawn thread IDs from one to the number of threads spawned minus one.

When the threads join, the original parent thread (spawn thread ID 0) leaves the join after all other threads that were spawned last have also joined. The join operation contains an implicit barrier, so that all threads must reach the join before the original thread continues.

Spawns and joins may thus be arbitrarily nested; however, each thread that was allocated as a result of a spawn must eventually join for correct program operation. The spawn thread ID has a scope between the immediately enclosing spawn/join pair; a single thread may change its spawn thread ID as the result of executing a spawn or join operation.

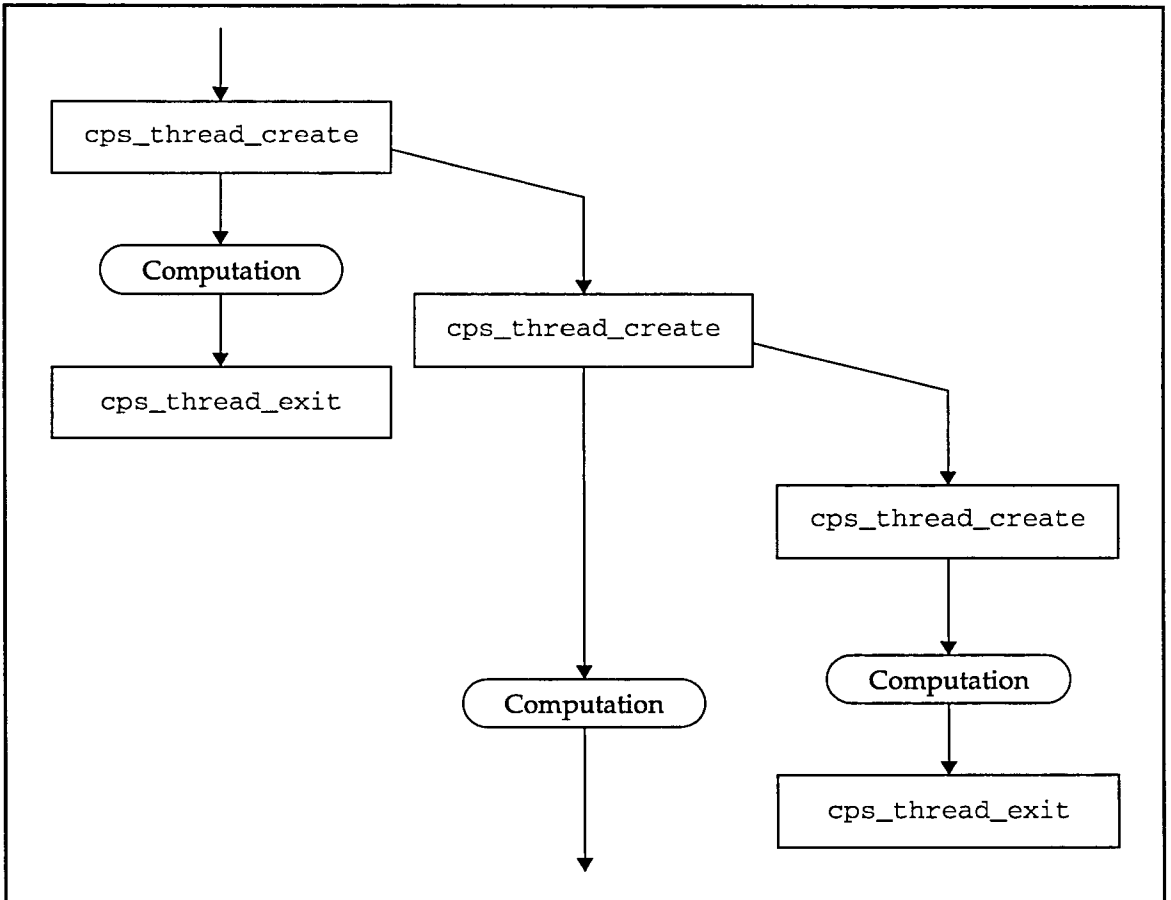
---

## Asymmetric Parallelism

Asymmetric parallelism is used when each thread executes a different, independent instruction stream. Asymmetric threads are analogous to the UNIX `fork` system call construct; the threads are disjoint. Either the parent or the child may terminate first in any order. Either or both the parent and the child may spawn additional symmetric or asymmetric threads.

Asymmetric threads are spawned using the `cps_thread_create()` function and terminated using the `cps_thread_exit()` function. Asymmetric threads can not join with their parent thread; they terminate separately. If you do not specifically call `cps_thread_exit()` to terminate an asymmetric thread, the thread will automatically terminate when the procedure called by `cps_thread_create()` successfully terminates.

Figure 27 shows an asymmetric thread tree. Asymmetrically spawned child threads do not have spawn thread IDs; they do, however, have unique kernel thread IDs, which are assigned in no particular order. The parent which executed the `cps_thread_create()` retains the same kernel thread ID it had before it spawned the child thread.



**Figure 27** Asymmetric parallelism

You can spawn symmetric threads from asymmetric threads using `cps_ppcall` and `cps_ppcalln`. In this case, the parent thread retains its kernel thread ID and, along with the symmetric threads, receives a spawn thread ID. These symmetric threads must join before the asymmetric parent can exit; if an asymmetric parent attempts to exit while its symmetric children are still active, it will join instead.

---

## Accessing CPSlib

C programs which use CPSlib functions must include the header file `cps.h`, as shown:

```
#include <cps.h>
```

To access `errno` symbolic constant values in C, you must also include the header file `errno.h`:

```
#include <errno.h>
```

C and Fortran share a common interface to CPSlib; the same library, `libcps.a`, allows access to the CPSlib functions from either language. This library is automatically linked when you compile your program with the Convex Exemplar `fc` or `cc` compiler drivers.

---

## CPS library functions

CPSlib provides thread management functions, high level synchronization functions, and low level synchronization functions. This section briefly describes each function and its arguments.

Default versions of the functions presented here return four byte values and take four byte arguments. Eight byte versions are also available; the names of these functions are suffixed with `_8` (e.g. the 8 byte version of `cps_ppcall` is `cps_ppcall_8`), and the functions take eight byte arguments. Refer to the appropriate man pages for more information.

All C examples presented here assume that `cps.h` is included in the program. Note that the `NULL` value in C is equivalent to 0 (zero) in Fortran.

## Note

**CPSlib routines are generally incompatible with system functions of the form `cnx_*`; mixing the two may cause wrong answers, deadlock or runtime errors. Mixing CPSlib routines and certain compiler directives may cause wrong answers at `-O1` or higher; if this happens, lower the optimization level or refer to the "Global register allocation" section of Chapter 3, "Compiler optimizations."**

---

## Thread management functions

These functions allow you to spawn and join or terminate threads.

### Symmetric thread functions

Symmetric threads are spawned, execute in parallel, and join via a single CPSlib function call. By definition, all parallel symmetric threads must execute to completion before the join operation can take place; therefore no `exit` or `wait` functions are provided.

## cps\_ppcall and cps\_ppcalln

The `cps_ppcall` and `cps_ppcalln` functions allow you to spawn symmetrically parallel threads. In Fortran, these functions have the following forms:

```
INTEGER FUNCTION CPS_PPCALL(PARAMS, FUNC, ARG)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                   !STRUCTURE IN C CODE
```

EXTERNAL FUNC

```
INTEGER FUNCTION CPS_PPCALLN(PARAMS, FUNC, n, ARG1, ..., ARGn)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                   !STRUCTURE IN C CODE
```

EXTERNAL FUNC

In C:

```
typedef struct {
    int node;          /* node to place allocated threads on */
    int min;          /* minimum number of threads to allocate */
    int max;          /* maximum number of threads to allocate */
    int threadscope; /* thread scope attributes */
} spawn_sym_t;

int cps_ppcall(spawn_sym_t *params, void (*func)(void *), void *arg);
int cps_ppcalln(spawn_sym_t *params, void (*func)(void *),
                const int *n, void *arg1, ..., void *argn);
```

The elements `params->node` and `PARAMS(1)` control what hypernodes threads are allocated on. They can contain the hypernode ID of the hypernode on which to allocate the threads, or they can take one of the values shown in Table 11.

**Table 11** `params->node`/`PARAMS(1)` values

C symbolic constant name	Value	Meaning
<code>CPS_SAME_NODE</code>	-1	Allocate threads on same hypernode as calling thread
<code>CPS_ANY_NODE</code>	-2	Allocate threads on any hypernode.
<code>CPS_DIFFERENT_NODE</code>	-3	Allocate threads on different hypernode than that of the calling thread.

`params->min` and `PARAMS(2)` specify the minimum number of threads to allocate; `params->max` and `PARAMS(3)` specify the maximum number of threads to allocate.

`params->threadscope` and `PARAMS(4)` control the creation strategy for new threads. Table 12 shows acceptable values for these parameters. Except where noted, the logical or of two or more of these values can be used.

**Table 12** `params->threadscope/PARAMS(4)` values

C symbolic constant name	Value	Meaning
<code>CPS_THREAD_PARALLEL</code>	1	Allocate multiple threads per hypernode. Mutually exclusive with <code>CPS_NODE_PARALLEL</code> .
<code>CPS_NODE_PARALLEL</code>	2	Allocate one thread per hypernode. Mutually exclusive with <code>CPS_THREAD_PARALLEL</code> .
<code>CPS_OVER_SUBSCRIBE</code>	4	Allows multiple threads per CPU. If not set, no more than one thread per CPU can be started.
<code>CPS_IGNORE_STACKSCOPE</code>	8	Allows the spawning of new threads which may not be able to validly address their parent's stack or <code>thread_private</code> data.

`cps_ppcall` spawns the number of threads designated in the argument `params` and arranges for each thread, including the calling thread, to call the argument function `func` with the argument `arg`. After returning from `func`, each thread automatically joins. When all threads have joined, the parent continues executing the code following the `cps_ppcall` or `cps_ppcalln`.

`cps_ppcalln` is identical to `cps_ppcall`, except that it will pass  $n$  arguments to the function `func`, where  $n$  ranges from 0 to 256.

The thread that calls `cps_ppcall` or `cps_ppcalln` is considered the parent thread; if the call is successful, it will become spawn thread 0 prior to calling the function. Other threads spawned will be assigned increasing spawn thread IDs ranging from 1 to  $m-1$ , where  $m$  is the number of threads spawned.

After all of the threads return from `func`, the parent thread will restore its previous thread state and continue execution after `cps_ppcall` or `cps_ppcalln`.

If successful, these functions return the number of threads spawned including the parent. If an error occurs, they return -1 and, in C, `errno` is set as shown in Table 13.

**Table 13** errno values for `cps_ppcall` and `cps_ppcalln`

errno value	Meaning
EAGAIN	The minimum number of threads could not be allocated.
EINVAL	Either the hypernode specified by <code>params-&gt;node</code> (or <code>PARAMS(1)</code> ) is not a valid logical hypernode ID, or the value of <code>params-&gt;min</code> ( <code>PARAMS(2)</code> ) or <code>params-&gt;max</code> ( <code>PARAMS(3)</code> ) is less than 0.

Nested `cps_ppcall` and/or `cps_ppcalln` calls are permitted.

## Note

Each thread spawned to run `func` receives a default local stack of 8 Mbytes. If `func` declares local variables that occupy more than 8 Mbytes, you must change this default using the `CPS_STACK_SIZE` environment variable. `CPS_STACK_SIZE` specifies the default stack size for spawned parallel functions in kbytes. It is read once at program startup; the spawn thread stack size cannot be changed during execution. Thread 0's default stack size is specified by `SPP-UX`; you can modify it via the `mpa(1)` utility.

For more information, refer to the `cps_ppcall(3)` man page.

### Asymmetric thread functions

By definition, asymmetric threads execute independently of one another. Rather than join, asymmetric threads *terminate*; that is, since their execution is independent, one asymmetric thread has no need to know when another has completed, so no join is necessary or possible.

#### `cps_thread_create` and `cps_thread_createn`

The `cps_thread_create` and `csp_thread_createn` functions allow you to spawn asymmetrically parallel threads. In Fortran, these functions have the following form:

```
INTEGER FUNCTION CPS_THREAD_CREATE(NODE, FUNC, ARG)
INTEGER NODE
EXTERNAL FUNC
```

```
INTEGER FUNCTION CPS_THREAD_CREATEN(NODE, FUNC, n, ARG1, ..., ARGn)
INTEGER NODE
EXTERNAL FUNC
```

In C:

```
int cps_thread_create(const int *node, void (*func) (void *),
                     void *arg);

int cps_thread_createn(const int *node, void (*func) (void*),
                      const int *n, void *arg1, ..., void *argn);
```

`cps_thread_create` spawns a single asymmetric thread on the hypernode specified by `node` and arranges for the asymmetric thread to call the argument function `func` with the parameter `arg`. On return from `func`, the asymmetric thread will automatically call `cps_thread_exit()`; this function can also be manually called from within `func` to terminate the asymmetric thread.

`node` takes the same values as the `node` argument to `cps_ppcall`, which is described in the “Symmetric thread functions” section.

`cps_thread_createn` is identical to `cps_thread_create`, except that it will pass `n` arguments to the function `func`, where `n` ranges from 0 to 256.

If successful, these functions return the kernel thread ID of the newly spawned thread. No assumptions can be made about kernel thread IDs except that they are unique. Asymmetric threads do not have spawn thread IDs.

## Caution

`cps_thread_create[n]` returns immediately after initiating `func`, and both `func` and the parent thread execute in parallel until one terminates. If the parent thread manipulates `arg` after returning from `cps_thread_create[n]` but before `func` exits, you must ensure that this manipulation is synchronized between the parent and `func`, or the value of `arg` will be indeterminate.

If unsuccessful, these functions return -1 and, in C, set `errno` as shown in Table 14.

**Table 14** `errno` values for `cps_thread_create[n]`

<code>errno</code> value	Meaning
EAGAIN	The asymmetric thread could not be allocated.
EINVAL	<code>node</code> is not a valid hypernode ID in the program-assigned subcomplex

Nested calls to `cps_thread_create` and/or `cps_thread_createn` are permitted.

For more information, refer to the `cps_thread_create(3)` man page.

### **cps\_thread\_exit**

This function terminates the asymmetric thread call made by `cps_thread_create`. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_EXIT()
```

In C:

```
int cps_thread_exit(void);
```

Upon successful completion of `func` (specified in `cps_thread_create`), asymmetric threads spawned with `cps_thread_create` will automatically call `cps_thread_exit`. The function is provided for cases in which you wish to terminate an asymmetric thread before `func` normally returns.

If successful, `cps_thread_exit` does not return. If unsuccessful it returns -1 and, in C, sets `errno` to `EINVAL` if it was called from a symmetric thread.

For more information, refer to the `cps_thread_create(3)` man page.

### **cps\_thread\_wait**

This function can be used to wait until all asymmetric threads have terminated, or to find out the number of active asymmetric threads. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_WAIT(FLAG)
```

In C:

```
int cps_thread_wait(const int *flag);
```

If `flag` is set, this routine waits until all asymmetric threads in the program have terminated. If `flag` is not set, it returns the number of active asymmetric threads in the program.

`cps_thread_wait` cannot be called with `flag` set from an asymmetric thread because the active state of the calling thread will prevent the function from returning, resulting in deadlock.

To use `cps_thread_wait` in an asymmetric thread to wait for all child asymmetric threads to terminate, you must know how many asymmetric threads were spawned before the calling thread. With this information, you can construct a loop which calls `cps_thread_wait` with `flag` equal to 0 until the number of active threads is equal to the number of previously spawned threads plus 1 (the calling thread), as shown in the following Fortran example:

```

10  IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
    IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
    IF(IWAIT .GT. PREVTHRDS+1) GOTO 10 ! SPIN UNTIL ALL
                                   ! CHILDREN TERMINATE

```

Here, `CPS_THREAD_WAIT` returns the total number of asymmetric threads at line 10; the next line is a routine error trap, and the third line checks the returned number against the known number of threads that are not children of the calling thread. `PREVTHRDS` is a user-defined and user-incremented variable. The loop cannot terminate until the returned number indicates that all of the calling thread's children have terminated.

If unsuccessful, `cps_thread_wait` returns -1 and, in C, sets `errno` to `EDEADLK` if it was called from an asymmetric thread or the child of an asymmetric thread with `flag` set.

For more information, refer to the `cps_thread_create(3)` man page.

### Thread information and attribute functions

These functions provide information on active parallel threads, the subcomplex configuration, and `cps` attributes. For information beyond that which follows, refer to the `cps_info(3)` man page.

#### `cps_stid`

This function returns the spawn thread ID of the calling thread. In Fortran `cps_stid` has the following form:

```
INTEGER FUNCTION CPS_STID()
```

In C:

```
stid_t cps_stid(void);
```

Spawn thread IDs range from 0.. $n-1$ , where  $n$  is the number of currently active symmetric threads in the current spawn context (refer to `cps_nsthreads`).

If unsuccessful, this function returns -1. Because asymmetric threads have no spawn thread IDs, `cps_stid()` returns -1 when called from an asymmetric thread.

#### `cps_ktid`

This function returns the kernel thread ID of the calling thread. In Fortran, `cps_ktid` has the following form:

```
INTEGER FUNCTION CPS_KTID()
```

In C:

```
tid_t cps_ktid(void);
```

Note that kernel threads IDs are generated with no regularity; they are simply unique IDs.

### **cps\_nsthreads**

This function returns the number of threads in the current *cps spawn context*. Each spawn establishes a spawn context and the number returned by *cps\_nsthreads* can vary from spawn to spawn. For example, if you have a *cps\_ppcall* which spawns 2 threads nested within a *cps\_ppcall* that spawns 4 threads, *cps\_nsthreads* returns 2 when called from the inner spawn context, and 4 when called from the outer spawn context.

In Fortran, *cps\_nsthreads* has the following form:

```
INTEGER FUNCTION CPS_NSTHREADS()
```

In C:

```
int cps_nsthreads(void);
```

### **cps\_plevel**

This function can be used to determine the current level of parallelism. In Fortran, it has the following form:

```
INTEGER FUNCTION CPS_PLEVEL()
```

In C:

```
int cps_plevel(void);
```

The return value is a bit mask. The possible return values are a sum of those shown in Table 15.

**Table 15** *cps\_plevel* return values

<b>C symbolic constant name</b>	<b>Value</b>	<b>Meaning</b>
CPS_PL_NONE	0	No parallel threads
CPS_PL_PARALLEL	1	Asymmetric parallel threads are active
CPS_PL_NODE	2	Hypernode-parallel threads are active
CPS_PL_NTHREAD	4	Parallel threads are active on the current hypernode.
CPS_PL_THREAD	8	Parallel threads are active across multiple hypernodes.
CPS_PL_ASYMMETRIC	16	Current thread is an asymmetric thread or a child of one.

**cps\_node\_id**

This function returns the logical ID of the hypernode on which the calling thread is executing. In Fortran, `cps_node_id` has the following form:

```
INTEGER CPS_NODE_ID()
```

In C:

```
int cps_node_id(void);
```

Logical hypernode IDs range from  $0..n-1$ , where  $n$  is the number of available hypernodes in the subcomplex. Logical IDs are assigned in the order in which your program occupies the subcomplex. The hypernode that your program's thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because SPP-UX starts a program on the least-loaded hypernode, logical hypernode IDs can differ between programs due to load balancing; thus two programs running on the same subcomplex are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs have no correlation to physical hypernode IDs, which are unique for each hypernode at the machine level.

**cps\_node\_cpus**

This function returns the number of CPUs available to the caller on the hypernode on which it is running. In Fortran it has the following form:

```
INTEGER CPS_NODE_CPUS()
```

In C:

```
int cps_node_cpus(void);
```

Note that the return value represents the number of CPUs visible to the calling application, and does not necessarily indicate the total number of physical CPUs on the hypernode.

**cps\_node\_nthreads**

This function returns the number of active threads belonging to the calling application that are running on the hypernode on which the call is executing. In Fortran it has the following form:

```
INTEGER CPS_NODE_NTHREADS()
```

In C:

```
int cps_node_nthreads(void);
```

Active threads include threads spawned by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads spawned automatically due to compiler-generated parallelism.

### **`cps_is_parallel`**

This function returns 1 if the program has parallel code and can go parallel; otherwise it returns 0. In Fortran it has the following form:

```
INTEGER CPS_IS_PARALLEL()
```

In C:

```
int cps_is_parallel(void);
```

Note that the `mpa` utility can be used to modify attributes of the executable such as whether or not it is parallel. Refer to the `mpa(1)` man page for more information.

### **`cps_complex_cpus`**

This function returns the total number of CPUs available to the application. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_CPUS()
```

In C:

```
int cps_complex_cpus(void);
```

### **`cps_complex_nthreads`**

This function returns the number of active threads belonging to the calling application that are running on the subcomplex on which the call is executing. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_NTHREADS()
```

In C:

```
int cps_complex_nthreads(void);
```

Active threads include threads spawned by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads spawned automatically due to compiler-generated parallelism.

### **`cps_complex_nodes`**

This function returns the total number of logical hypernodes available to the application from which it is called in the application's subcomplex. In Fortran it has the following form:

```
INTEGER CPS_COMPLEX_NODES()
```

In C:

```
int cps_complex_nodes(void);
```

**cps\_topology**

This function fills an array with the application's view of the topology of the subcomplex on which it is running. In Fortran it has the following form:

```
INTEGER CPS_TOPOLOGY (NODES, N)
INTEGER NODES (N)
```

In C:

```
int cps_topology(int nodes[], int n)
```

Each element of the `NODES` or `nodes` array corresponds to a logical hypernode, with the first element corresponding to hypernode 0, the second to hypernode 1, etc. Therefore, given default subscripting in Fortran, `NODES(1)` represents hypernode 0. Given default subscripting in C, `nodes[0]` corresponds to hypernode 0. On return from `cps_topology`, each element contains the number of threads running on the corresponding hypernode that belong to the calling application.

`N` or `n` represents the number of elements in the array. If there are more elements than actual hypernodes, the remaining elements are set to 0.

**cps\_wait\_attr**

As described in Chapter 3, "Compiler optimizations," idle threads can either be suspended or spin-waiting. This function allows you to get or change the current CPSlib thread wait attributes. In Fortran it has the following form:

```
INTEGER CPS_WAIT_ATTR (CMD, WAIT)
INTEGER CMD, WAIT(2)
```

In C:

```
typedef struct {
    int wait_attr; /* idle wait attribute */
    int wait_time; /*wait time in milliseconds*/
} cps_spin_state_t;
int cps_wait_attr(int *cmd, cps_wait_attr_t
                 *wait);
```

Where `CMD` or `cmd`, depending on its contents, indicates whether to get or change the current wait attributes. Accepted values are shown in Table 16.

**Table 16** Accepted CMD/cmd values

<b>C symbolic constant name</b>	<b>Value</b>	<b>Meaning</b>
CPS_GETWAIT	0	Get the current wait attribute values and store them in the WAIT array or in the structure pointed to by wait.
CPS_SETWAIT	1	Set the wait attributes to the values in the WAIT array or pointed to by wait. These values become effective after the next join or cps_thread_exit is executed.
CPS_SETWAITI	2	Sets the wait attributes as CPS_SETWAIT does, but forces all active threads to join immediately.

The WAIT array or the structure pointed to by wait contain the new or current wait attributes, depending on the value of cmd. WAIT(1) and wait->wait\_attr take one of the values shown in Table 17.

**Table 17** WAIT(2)/wait->wait\_attr values

<b>C symbolic constant name</b>	<b>Value</b>	<b>Meaning</b>
CPS_SUSPEND	1	Suspend the thread after waiting the time specified in WAIT(2) or wait->wait_time.
CPS_SPINWAIT	2	Spin-wait until the thread is reactivated.

WAIT(1) and wait->wait\_time take an integer representing the number of milliseconds the thread is to spin-wait before suspending itself (assuming the wait attribute is set to CPS\_SUSPEND). These values default to 10 ms for non-oversubscribed threads and 0 ms for oversubscribed threads.

If unsuccessful, cps\_wait\_attr returns -1 and, in C, sets errno to EINVAL if:

- wait\_time is greater than CPS\_WAIT\_MAX, and wait\_attr is set to CPS\_SUSPEND
- wait\_attr is not set to CPS\_SUSPEND or CPS\_SPINWAIT or
- cmd is not CPS\_GETWAIT, CPS\_SETWAIT, or CPS\_SETWAITI

## High-level synchronization functions

These routines manipulate the barriers and mutexes which are used for synchronization. CPSlib barriers can be used to construct barriers such as those that can be constructed with compiler directives as described in Chapter 6, "Advanced shared memory programming." Mutexes are areas of mutual exclusion, and are analogous to directive-controlled critical sections described in Chapter 6.

These constructs offer a lower degree of automation and a higher degree of control than those directive-specified constructs described in Chapter 6. However, they offer a higher degree of automation and a lower degree of control than the constructs that can be manually built using low-level synchronization functions described in the following section.

All of the functions described in this section return 0 on success and -1 on failure.

For information beyond that which follows, refer to the `cps_barrier(3)` and `cps_mutex(3)` man pages.

### `cps_barrier_alloc`

This function allocates the barrier `barr` and sets its associated shared counter to zero. When each thread reaches the barrier, it increments the counter; when the counter equals the number of parallel threads, all threads may proceed.

In Fortran this function has the form:

```
INTEGER FUNCTION CPS_BARRIER_ALLOC(BARR)
INTEGER BARRIER
```

In C:

```
int cps_barrier_alloc(barrier_t *barr);
```

If this function fails when called from C, `errno` is set to `ENOMEM` if the memory required for `barr` cannot be allocated.

### `cps_barrier_free`

This function releases the barrier `barr`. In Fortran it has the form:

```
INTEGER FUNCTION CPS_BARRIER_FREE(BARR)
INTEGER BARR
```

In C:

```
int cps_barrier_free(barrier_t *barr);
```

If this function fails when called from C, `errno` is set to `EINVAL` if `barr` was not allocated with a CPSlib barrier allocation

function, or to EBUSY if the counter associated with `barr` is nonzero.

### **cps\_barrier**

This function increments the shared counter associated with the barrier `barr`. When the value of the shared counter is equal to the argument `nthreads`, the function returns, and the counter is set to zero. In Fortran, `cps_barrier` has the following form:

```
INTEGER FUNCTION CPS_BARRIER (BARR, NTHREADS)
INTEGER BARR, NTHREADS
```

In C:

```
int cps_barrier(barrier_t *barr, const int
                *nthreads);
```

If this function fails when called from C because `barr` was not allocated with a CPSlib barrier allocation function, `errno` is set to `EINVAL`.

### **cps\_mutex\_alloc**

This function allocates the mutex `mutex` and unlocks it. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_ALLOC (MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_alloc(cps_mutex_t *mutex);
```

This function does not check whether `mutex` is already allocated; therefore, when successful, it always allocates a new mutex. When it fails, `mutex` is set to `NULL` (in Fortran, `MUTX` is set to 0).

If this function fails when called from C, it sets `errno` to `ENOMEM` if it cannot allocate the required memory.

### **cps\_mutex\_free**

This function releases the mutex `mutex`. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_FREE (MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_free(cps_mutex_t *mutex);
```

If this function is successful when called from C, `mutex` is set to `NULL`. In Fortran, `MUTX` is undefined.

If unsuccessful when called from C, it sets `errno` to `EINVAL` if `mutex` was not allocated by a CPSlib allocation function, or to `EBUSY` if `mutex` has already been acquired.

**cps\_mutex\_lock**

If the mutex `mutex` is unlocked, this function acquires it and returns; if it is locked by another thread, `cps_mutex_lock` will wait until it is acquired before returning.

In Fortran, `cps_mutex_lock` has the following form:

```
INTEGER FUNCTION CPS_MUTEX_LOCK (MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_lock(cps_mutex_t *mutex);
```

If the calling thread has already acquired `mutex` this function returns -1 and, in C, sets `errno` to `EDEADLK`.

**cps\_mutex\_unlock**

This function releases the mutex `mutex` so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_MUTEX_UNLOCK (MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_unlock(cps_mutex_t *mutex);
```

**cps\_mutex\_trylock**

This function attempts to acquire `mutex`; if `mutex` is already locked by another thread, `cps_mutex_trylock` will return -1.

In Fortran, `cps_mutex_trylock` has the following form:

```
INTEGER FUNCTION CPS_MUTEX_TRYLOCK (MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_trylock(cps_mutex_t *mutex);
```

If the calling thread has already acquired `mutex` this function returns -1 and, in C, sets `errno` to `EDEADLK`.

**Low-level synchronization functions**

These functions manipulate the counters and semaphores used for low-level synchronization. These functions require you to create and manually control your own synchronization semaphores.

Semaphores can be cache- or memory-based. Cache-based semaphores can be stored in the processor data cache, making them best for use in situations that generate minimal semaphore contention. Memory based semaphores are never brought into the

processor data cache, making them preferable in situations that generate semaphore contention.

Because each semaphore has an associated counter, it can be used both as a lock (to implement, for example, a critical section) and a counter (to implement, for example, an ordered section).

For information beyond that which follows, refer to the `cps_sema(3)` man page.

### **c\_init32**

This function allocates the cache-based semaphore `cs` and initializes its associated counter to `value`. In Fortran it has the following form:

```
INTEGER FUNCTION C_INIT32(CS, VALUE)
INTEGER CS, VALUE
```

In C:

```
int c_init32(mem_sema_t *cs, const int *value);
```

If successful, `c_init32` returns the counter value; otherwise it returns -1.

### **c\_free32**

This function frees the cache-based semaphore `cs` and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION C_FREE32(CS)
INTEGER CS
```

In C:

```
int c_free32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

### **c\_lock**

This function acquires a cache-based semaphore `cs`. In Fortran it has the following form:

```
INTEGER FUNCTION C_LOCK(CS)
INTEGER CS
```

In C:

```
int c_lock(cache_sema_t *cs);
```

If the semaphore is already acquired, `c_lock` will wait until the semaphore is released before returning. It may or may not give up the processor in the interim.

**c\_unlock**

This function releases the cache-based semaphore *cs* so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION C_UNLOCK (CS)
INTEGER CS
```

In C:

```
int c_unlock(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

**c\_cond\_lock**

If the cache-based semaphore *cs* is available, this function acquires it; otherwise, it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION C_COND_LOCK (CS)
INTEGER CS
```

In C:

```
int c_cond_lock(cache_sema_t *cs);
```

**c\_fetch32**

This function returns the value of the counter associated with the cache-based semaphore *cs*. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH32 (CS)
INTEGER CS
```

In C:

```
int c_fetch32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

**c\_fetch\_and\_inc32**

This function increments the value of the counter associated with the semaphore *cs* and returns the old value.

In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_INC32 (CS)
INTEGER CS
```

In C:

```
int c_fetch_and_inc32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

### **c\_fetch\_and\_dec32**

This function decrements the value of the counter associated with the semaphore `cs` and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_DEC32 (CS)
INTEGER CS
```

In C:

```
int c_fetch_and_dec32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

### **c\_fetch\_and\_clear32**

This function returns the current value of the counter associated with the semaphore `cs` and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_CLEAR32 (CS)
INTEGER CS
```

In C:

```
int c_fetch_and_clear32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

### **c\_fetch\_and\_add32**

This function adds `value` to the counter associated with the semaphore `cs` and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_ADD32 (CS, VALUE)
INTEGER CS, VALUE
```

In C:

```
int c_fetch_and_add32(cache_sema_t *cs, const
                    int *value);
```

If unsuccessful, this function returns -1.

### **c\_fetch\_and\_set32**

This function returns the current value of the counter associated with the semaphore `cs`, and sets the semaphore to the new value contained in `newval`. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_SET32 (CS, NEWVAL)
INTEGER CS, NEWVAL
```

In C:

```
int c_fetch_and_set32(cache_sema_t *cs, const
                    int *newval);
```

If unsuccessful, this function returns -1.

### **m\_init32**

This function allocates the memory-based semaphore `ms` and initializes the counter associated with it to `value`. In Fortran it has the following form:

```
INTEGER FUNCTION M_INIT32 (MS, VALUE)
INTEGER MS, VALUE
```

In C:

```
int m_init32(mem_sema_t *ms, const int *value);
```

If successful, this function returns the counter value; otherwise it returns -1.

### **m\_free32**

This function releases the memory-based semaphore `ms` and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION M_FREE32 (MS)
INTEGER MS
```

In C:

```
int m_free32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

### **m\_lock**

This function acquires the memory-based semaphore `ms`. If the semaphore is already acquired, `m_lock` will wait until the semaphore is released before returning. It may or may not give up the processor in the interim. In Fortran it has the following form:

```
INTEGER FUNCTION M_LOCK (MS)
INTEGER MS
```

In C:

```
int m_lock(mem_sema_t *ms);
```

### **m\_unlock**

This function releases the memory-based semaphore `ms` so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION M_UNLOCK (MS)
INTEGER MS
```

In C:

```
int m_unlock(mem_sema_t *ms)
```

If unsuccessful, this function returns -1.

### **m\_cond\_lock**

If the memory-based semaphore *ms* is available, this function acquires it; otherwise it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION M_COND_LOCK (MS)
INTEGER MS
```

In C:

```
int m_cond_lock(mem_sema_t *ms);
```

### **m\_fetch32**

This function returns the value of the counter associated with the memory-based semaphore *ms*. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH32 (MS)
INTEGER MS
```

In C:

```
int m_fetch32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

### **m\_fetch\_and\_inc32**

This function increments the value of the counter associated with the semaphore *ms* and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_INC32 (MS)
INTEGER MS
```

In C:

```
int m_fetch_and_inc32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

### **m\_fetch\_and\_dec32**

This function decrements the value of the counter associated with the semaphore *ms* and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_DEC32 (MS)
INTEGER MS
```

In C:

```
int m_fetch_and_dec32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

**m\_fetch\_and\_clear32**

This function returns the current value of the counter associated with the semaphore *ms* and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_CLEAR32 (MS)
INTEGER MS
```

In C:

```
int m_fetch_and_clear32 (mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

**sync\_routine directive and pragma**

Among the most basic optimizations performed by the Convex SPP Series compiler is code motion, which is described in Chapter 3, "Compiler optimizations." This optimization can move some code across function calls. If the function call is to a synchronization or parallelization function and the code moved must execute on a certain side of it, this movement can cause wrong answers. Anytime you use CPSlib functions rather than the directives or functions described in Chapter 4, "Basic shared memory programming," and Chapter 6, "Advanced shared memory programming," to synchronize or parallelize code, you must identify the functions with a *sync\_routine* directive or pragma. *sync\_routine* should be used to identify all CPSlib functions, as well as any user-written functions that accomplish synchronization or parallelization or hide calls to any synchronization or parallelization functions.

In Fortran, *sync\_routine* has the following form:

```
C$DIR SPP SYNC_ROUTINE (funclist)
```

In C, it has the following form:

```
#pragma _CNX SPP sync_routine (funclist)
```

Where *funclist* is a comma-delimited list of synchronization procedures.

*sync\_routine* is only effective for the listed procedures which lexically follow it in the procedure in which it appears.

Consider the following Fortran example:

```
SUBROUTINE WORK(ARG1, ARG2, MUTX)
  INTEGER ARG1, ARG2, MUTX, CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
  C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
  .
  .
  .
  DO I = 1, N
    .
    .
    .
    LCK = CPS_MUTEX_LOCK(MUTX)
    .
    .
    .
    LCK = CPS_MUTEX_UNLOCK(MUTX)
  ENDDO
  .
  .
  .
  END
```

Here, the subroutine `WORK` is called in parallel and contains a loop which contains a critical section protected by calls to CPSlib functions. Listing these CPSlib functions in a `SYNC_PARALLEL` directive at the beginning of the subroutine prevents the compiler from moving code out of the critical section.

An analogous C example follows:

```
#include <spp_prog_model.h>
work(int arg1, int arg2, int mutx) {
  int i, lck;
  #pragma _CNX sync_routine(cps_mutex_lock, cps_mutex_unlock)
  .
  .
  .
  #pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
    .
    .
    .
    lck = cps_mutex_lock(&mutx);
    .
    .
    .
    lck = cps_mutex_unlock(&mutx);
  }
}
```

---

## Examples

The examples presented here demonstrate various constructs that can be programmed using the CPSlib functions described in the previous sections.

You can compile a program that uses CPSlib to achieve parallelism at any optimization level, however, you must pass appropriate flags to the linker to indicate that the program is parallel. This is done using the `-Wl`, option, which indicates that the comma-delimited options following it up to the next space are to be passed to the linker. You can pass either the `+min` and `+max` options to indicate the minimum and maximum number of CPUs on which to run the program, or the `+parallel` option to indicate that the program is parallel and that any available CPUs should be used.

The following example `fc` command line compiles the program `cpspar.f` at optimization level `-O2` and indicates to the linker that the program should run on a minimum of 2 and a maximum of 8 processors:

```
fc -Wl,+min,2,+max,8 -O2 cpspar.f
```

The compiler does not recognize CPSlib parallelism, so while this command line would generate an optimization report, the report would only detail compiler optimizations; it would not report manually-coded parallelism. To see if your executable is parallel, use the `file` command.

You can also use the `mpa` utility to modify many attributes, including parallelism, of an executable file after compilation. Refer to the `mpa(1)` man page for more information.

---

## Symmetric parallelism

There are two forms of symmetric parallelism: block parallelism, and cyclic parallelism. The CPSlib functions used in the examples that follow are described in detail in the “CPS library functions” section.

### Block parallelism

Block parallelism is the most commonly used form of parallelism; it is the form generated by default by Convex SPP Series compilers. It involves splitting up the iterations of a loop into iteration blocks of similar size, and running each block on a separate processor. Block parallelism is illustrated in Chapter 3, “Compiler optimizations,” starting on page 96.

A simple Fortran example which uses CPSlib to implement block parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```

PROGRAM CPSBLOCK
REAL X(1000), Y(1000), Z(1000)
INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
EXTERNAL PARBLK ! REQUIRED BECAUSE PARBLK IS AN ARGUMENT
C INITIALIZE PARGS ARRAY:
  PARGS(1) = -2 ! ALLOCATE THREADS ON CALLING THREAD'S NODE
  PARGS(2) = 2 ! MINIMUM OF 2 THREADS
  PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
  PARGS(4) = 1 ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
  ITHREAD = CPS_PPCALLN(PARGS, PARBLK, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
  IF (IThread .LT. 0) PRINT *, 'PPCALLN FAILED'
  .
  . ! SERIAL CODE
  .
END

SUBROUTINE PARBLK (X, Y, Z, NTHR)
REAL X(1000), Y(1000), Z(1000)
INTEGER CPS_NSTHEADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHEADS, CPS_STID)
STID = CPS_STID() ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED
ITPERPROC = 1000/NTHR ! COMPUTE ITERATIONS PER THREAD
IEXCESS = 1000-ITPERPROC*NTHR ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DON'T HANDLE EXCESS:
  IF(STID .GE. IEXCESS) THEN
    MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
    MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
  ELSE
    MYSTART = STID * (ITPERPROC+1) + 1
    MYEND = MYSTART + (ITPERPROC+1) - 1
  ENDIF
C ACTUAL COMPUTATION:
  DO J = MYSTART, MYEND
    Z(J) = X(J) + Y(J)
  ENDDO
  RETURN
END

```

This example calls `CPS_NODE_CPUS` to find the number of available processors, then calls `CPS_PPCALLN` to spawn parallel threads to run the subroutine `PARBLK`.

`PARBLK` then determines the number of iterations necessary per processor; if the number of processors does not integrally divide the number of iterations, it automatically adjusts some blocks to handle the extra iterations. Finally, the loop in `PARBLK` performs its body in parallel, with each thread operating on the appropriate iteration range.

Note the error trap immediately after the call to `CPS_PPCALLN`; this is important, as it provides the only means of knowing if the spawn failed.

### Cyclic parallelism

Cyclic parallelism distributes consecutive iterations of a loop to separate processors. It is similar to the parallelism achieved through use of the `loop_parallel (ordered)` directive and pragma, but it does not order the iterations automatically; you must handle any necessary ordering manually.

`loop_parallel (ordered)` is discussed in Chapter 6, "Advanced shared memory programming," on page 190.

A simple Fortran example which uses `CPSlib` to implement cyclic parallelism follows. The `CPSlib` functions used here are described in detail in the "CPS library functions" section.

```
PROGRAM CPSCYCLE
  REAL X(1000), Y(1000), Z(1000), SUM
  INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
  EXTERNAL PARCYC ! PARCYC IS AN ARGUMENT
  READ *, NPROCS
C INITIALIZE PARGS ARRAY:
  PARGS(1) = -2 ! ALLOCATE THREADS ON CALLING THREAD'S NODE
  PARGS(2) = 2 ! MINIMUM OF 2 THREADS
  PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
  PARGS(4) = 1 ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
  ITHREAD = CPS_PPCALLN (PARGS, PARCYC, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
  IF (ITHREAD .LT. 0) PRINT *, 'PPCALLN FAILED'
  .
  . ! SERIAL CODE
  .
END
```

```

SUBROUTINE PARCYC (X, Y, Z, NTHR)
REAL X(1000), Y(1000), Z(1000)
INTEGER CPS_NSTHEADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHEADS, CPS_STID)
STID = CPS_STID()      ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED
C ACTUAL COMPUTATION:
DO J = 1+STID, 1000, NTHR ! STEP BY NUMBER OF THREADS
    Z(J) = X(J) + Y(J)
ENDDO
RETURN
END

```

This example works exactly like the block-parallelism example, except the loop in PARCYC, its parallel subroutine, is cyclically parallel. Cyclic parallelism is accomplished here by offsetting the loop start value by spawn thread ID and stepping the loop by the number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop; NTHR elements are computed per step. Contiguous STIDs compute contiguous elements.

---

## Asymmetric parallelism

A simple Fortran program that implements asymmetric parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```

PROGRAM ASYM
REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
COMMON /POINTS/ X1, X2, Y1, Y2
EXTERNAL DISTANCE ! DISTANCE IS AN ARGUMENT
.
. ! SERIAL CODE
. ! EXAMPLE CONTINUED

```

```

C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE DISTANCE:
  ITHREAD = CPS_THREAD_CREATE(-2, DISTANCE, Z)
  IF (ITHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN MAIN"
  .
  . ! THIS CODE RUNS IN PARALLEL WITH DISTANCE
  .
  IWAIT = CPS_THREAD_WAIT(1) ! WAIT FOR ALL ASYMMETRIC
    ! THREADS TO TERMINATE
  IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
  .
  . ! THIS CODE RUNS SERIALLY AFTER PARALLEL THREADS
  . ! TERMINATE
  END

  SUBROUTINE DISTANCE(Z)
  REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
  REAL X3(1000), Y3(1000)
  INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
  COMMON /POINTS/ X1, X2, Y1, Y2
  EXTERNAL FINDX ! FINDX IS AN ARGUMENT
C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE FINDX:
  JTHREAD = CPS_THREAD_CREATE(-2, FINDX, X3)
  IF (JTHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN DISTANCE"
  DO I = 1, 1000 ! COMPUTE Y3 IN PARALLEL WITH FINDX
    Y3(I) = (Y2(I) - Y1(I))**2
  ENDDO
10  IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
  IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
  IF(IWAIT .GT. 1) GOTO 10 ! SPIN UNTIL ONLY THIS THREAD
    ! IS ACTIVE
  DO I = 1, 1000 ! COMPUTE Z SERIALLY AFTER X3 AND Y3
    Z(I) = SQRT(X3(I) + Y3(I))
  ENDDO
  RETURN
  END

  SUBROUTINE FINDX(X3) ! RUNS IN PARALLEL WITH COMPUTATION
    ! OF Y3
  REAL X1(1000), X2(1000), Y1(1000), Y2(1000), X3(1000)
  COMMON /POINTS/ X1, X2, Y1, Y2
  DO I = 1, 1000
    X3(I) = (X2(I) - X1(I))**2
  ENDDO
  RETURN
  END

```

In this example, the arrays X3 and Y3 must be computed before the array Z can be computed. The main program spawns an asymmetric parallel thread to run DISTANCE, which spawns an asymmetric thread to run FINDX. DISTANCE then computes Y3 while FINDX computes X3; all the while, the main program can be doing other work in parallel with both subroutines. When DISTANCE is done with Y3, it waits until FINDX is done with X3, then computes Z. The main program waits until DISTANCE is done, then proceeds with more work.

Note the way in which CPS\_THREAD\_WAIT is used when called from DISTANCE; this is explained further in the "CPS library functions" section.

---

## Synchronization using high-level functions

This section demonstrates how to use barriers and mutexes to synchronize symmetrically parallel code.

### Barriers

Remember that, when you use `cps_ppcall()` to spawn symmetric parallelism, before the function returns, a join operation takes place after all spawned threads terminate. This join is an implicit barrier, since thread 0 cannot proceed until all parallel threads terminate. In many cases, this is the only barrier synchronization you will require.

However, the `cps_barrier` high-level synchronization functions allow you to explicitly create barriers if necessary.

The following Fortran example is similar to the symmetric parallelism example on page 328 except that instead of relying on the implicit barrier contained in the call to `CPS_PPCALL()`, it contains an explicit `CPS_BARRIER()` in the subroutine SUMMER.

```
PROGRAM BAR
REAL A(1000)
REAL SUM(:), TOTSUM
INTEGER PARGS(4), SUMBAR, CPS_NODE_CPUS, CPS_PPCALLN
INTEGER CPS_BARRIER_ALLOC, CPS_BARRIER_FREE
C$DIR SYNC_ROUTINE(CPS_BARRIER_ALLOC,CPS_BARRIER_FREE)
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS,CPS_PPCALLN)
EXTERNAL SUMMER
ALLOCATABLE(SUM)
NCPUS = CPS_NODE_CPUS()
ALLOCATE(SUM(0:NCPUS-1)) ! ONE ELEMENT FOR EACH CPU
```

```

PARGS(1) = -2 ! ALLOCATE THREADS ON CALLING THREAD'S NODE
PARGS(2) = 2 ! MINIMUM OF 2 THREADS PER NODE
PARGS(3) = NCPUS ! MAXIMUM # OF THREADS PER NODE
PARGS(4) = 1 ! ALLOCATE MULTIPLE THREADS PER NODE
DO I = 0, NCPUS-1 ! INITIALIZE SUM
  SUM(I) = 0.0
ENDDO

.
. ! SERIAL CODE
.

IERR = CPS_BARRIER_ALLOC(SUMBAR) ! ALLOCATE BARRIER
IF (IERR .LT. 0) PRINT*, "BARRIER ALLOCATION FAILED"
C SPAWN PARALLEL THREADS:
IERR = CPS_PPCALLN(PARGS, SUMMER, 5, A, SUM, TOTSUM, SUMBAR, NCPUS)
IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
IERR = CPS_BARRIER_FREE(SUMBAR) ! FREE BARRIER
IF (IERR .LT. 0) PRINT*, "BARRIER FREE FAILED"

.
. ! SERIAL CODE
.
END

SUBROUTINE SUMMER(A, SUM, TOTSUM, SUMBAR, NCPUS)
INTEGER STID, NTHR, SUMBAR
INTEGER CPS_STID, CPS_NSTHEADS, CPS_BARRIER
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHEADS, CPS_BARRIER)
REAL A(1000), SUM(0:NCPUS-1), TOTSUM
STID = CPS_STID() ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED
ITPERPROC = 1000/NTHR ! COMPUTE ITERATIONS PER THREAD
IEXCESS = 1000-ITPERPROC*NTHR ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DON'T HANDLE EXCESS:
IF(STID .GE. IEXCESS) THEN
  MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
  MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
ELSE
  MYSTART = STID * (ITPERPROC+1) + 1
  MYEND = MYSTART + (ITPERPROC+1) - 1
ENDIF
C ACTUAL COMPUTATION:
DO J = MYSTART, MYEND ! EACH THREAD COMPUTES LOCAL SUM
  SUM(STID) = SUM(STID) + A(J)
ENDDO

```

```

C WAIT UNTIL ALL THREADS ARE DONE COMPUTING THEIR PORTION OF SUM:
  IERR = CPS_BARRIER(SUMBAR, NTHR)
  IF (IERR .LT. 0) PRINT*, "BARRIER FAILED"
  IF (STID .EQ. 0) THEN ! THREAD 0 COMPUTES TOTAL SUM
    DO I = 0, NTHR-1
      TOTSUM = TOTSUM + SUM(I)
    ENDDO
  ENDDIF
RETURN
END

```

Here, the subroutine SUMMER is called in parallel to compute the sum of the elements of array A. Each parallel thread computes its own sum in an element of the array SUM. The CPS\_BARRIER function is used to prevent execution of any further code until all threads have finished computing their portion of SUM. When CPS\_BARRIER returns, thread 0 computes TOTSUM, and SUMMER returns.

### Mutexes

CPSlib mutexes allow you to limit access to the sections of code they delimit to one thread at a time, allowing you to construct critical sections similar to those discussed in Chapter 4, "Basic shared memory programming."

In the following Fortran example, the routine SUMMER performs the same task it did in preceding barrier example. However, here access to the TOTSUM computation takes place in fully parallel code; it is limited to one thread at a time by the mutex SUMMUTEX. This eliminates the need for each thread to compute independent SUM arrays as in the preceding barrier example.

```

PROGRAM MUT
REAL A(1000)
REAL TOTSUM
INTEGER PARGS(4), SUMMUTEX
INTEGER CPS_NODE_CPUS, CPS_PPCALLN
INTEGER CPS_MUTEX_ALLOC, CPS_MUTEX_FREE
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS, CPS_PPCALLN)
C$DIR SYNC_ROUTINE(CPS_MUTEX_ALLOC, CPS_MUTEX_FREE)
EXTERNAL SUMMER
NCPUS = CPS_NODE_CPUS()
PARGS(1) = -2 ! ALLOCATE THREADS ON CALLING THREAD'S NODE
PARGS(2) = 2 ! MINIMUM OF 2 THREADS PER NODE
PARGS(3) = NCPUS ! MAXIMUM # OF THREADS PER NODE
PARGS(4) = 1 ! ALLOCATE MULTIPLE THREADS PER NODE

```

```

TOTSUM = 0.0          ! INITIALIZE TOTSUM
.
.   ! SERIAL CODE
.
IERR = CPS_MUTEX_ALLOC(SUMMUTEX) ! ALLOCATE MUTEX
IF (IERR .LT. 0) PRINT*, "MUTEX ALLOCATION FAILED"
C SPAWN PARALLEL THREADS:
IERR = CPS_PPCALLN(PARGS, SUMMER, 3, A, TOTSUM, SUMMUTEX)
IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
IERR = CPS_MUTEX_FREE(SUMMUTEX)  ! FREE MUTEX
IF (IERR .LT. 0) PRINT*, "MUTEX FREE FAILED"
.
.   ! SERIAL CODE
.
END

SUBROUTINE SUMMER(A, TOTSUM, SUMMUTEX)
INTEGER STID, NTHR, SUMMUTEX
INTEGER CPS_STID, CPS_NSTHEADS
INTEGER CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHEADS)
C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
REAL A(1000), TOTSUM
STID = CPS_STID()      ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED
ITPERPROC = 1000/NTHR ! COMPUTE ITERATIONS PER THREAD
IEXCESS = 1000-ITPERPROC*NTHR ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DON'T HANDLE EXCESS:
IF(STID .GE. IEXCESS) THEN
    MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
    MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
ELSE
    MYSTART = STID * (ITPERPROC+1) + 1
    MYEND = MYSTART + (ITPERPROC+1) - 1
ENDIF
C ACTUAL COMPUTATION:
DO J = MYSTART, MYEND
C MUTEX LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
IERR = CPS_MUTEX_LOCK(SUMMUTEX)
IF (IERR .LT. 0) PRINT*, "MUTEX LOCK FAILED"
TOTSUM = TOTSUM + A(J)
IERR = CPS_MUTEX_UNLOCK(SUMMUTEX)
IF (IERR .LT. 0) PRINT*, "MUTEX UNLOCK FAILED"
ENDDO
RETURN
END

```

Here, as in the barrier example, `SUMMER` is called in parallel. Each parallel thread then waits until it can lock `SUMMUTEX` before updating `TOTSUM`.

---

## Synchronization using low-level functions

This section demonstrates how to use semaphores to synchronize symmetrically parallel code.

### Critical sections

Critical sections like the one in the preceding mutex example can be implemented in a similar fashion using cache-based or memory-based semaphores.

The following Fortran example is identical to the mutex example, but implements the critical section using a memory-based semaphore instead of a mutex.

```
PROGRAM SEM
REAL A(1000)
REAL TOTSUM
INTEGER PARGS(4), SUMSEM, SEMCNT
INTEGER CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32)
EXTERNAL SUMMER
NCPUS = CPS_NODE_CPUS()
PARGS(1) = -2      ! ALLOCATE THREADS ON CALLING THREAD'S NODE
PARGS(2) = 2      ! MINIMUM OF 2 THREADS PER NODE
PARGS(3) = NCPUS  ! MAXIMUM # OF THREADS PER NODE
PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER NODE
TOTSUM = 0.0      ! INITIALIZE TOTSUM
SEMCNT = 0 ! COUNTER FOR SEMAPHORE; VALUE IS IRRELEVANT
.
. ! SERIAL CODE
.
IERR = M_INIT32(SUMSEM,SEMCNT) ! ALLOCATE SUMSEM
IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
IERR = CPS_PPCALLN(PARGS,SUMMER,3,A,TOTSUM,SUMSEM)
IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
IERR = M_FREE32(SUMSEM)
IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
.
. ! SERIAL CODE
.
END
```

```

SUBROUTINE SUMMER(A,TOTSUM,SUMSEM)
INTEGER STID, NTHR, SUMSEM
INTEGER CPS_STID, CPS_NSTHEADS, M_LOCK, M_UNLOCK
REAL A(1000),TOTSUM
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHEADS, M_LOCK, M_UNLOCK)
STID = CPS_STID()      ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED
ITPERPROC = 1000/NTHR ! COMPUTE ITERATIONS PER THREAD
IEXCESS = 1000-ITPERPROC*NTHR ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DON'T HANDLE EXCESS:
  IF(STID .GE. IEXCESS) THEN
    MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
    MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
  ELSE
    MYSTART = STID * (ITPERPROC+1) + 1
    MYEND = MYSTART + (ITPERPROC+1) - 1
  ENDIF
C ACTUAL COMPUTATION:
  DO J = MYSTART, MYEND
C SEMAPHORE LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
    IERR = M_LOCK(SUMSEM)
    IF (IERR .LT. 0) PRINT*, "SEMAPHORE LOCK FAILED"
    TOTSUM = TOTSUM + A(J)
    IERR = M_UNLOCK(SUMSEM)
    IF (IERR .LT. 0) PRINT*, "SEMAPHORE UNLOCK FAILED"
  ENDDO
RETURN
END

```

Here as in the mutex example, SUMMER is called in parallel. Each parallel thread then waits until it can lock the memory-based semaphore SUMSEM before updating TOTSUM.

### Ordered sections

Semaphores can also be used to construct ordered sections such as those constructed using the `loop_parallel(ordered)`, `begin_ordered_section` and `end_ordered_section` directives and pragmas, which are described in Chapter 6, "Advanced shared memory programming."

The parallel loop in the following Fortran example contains a backward-LCD, which is isolated using low-level synchronization functions so that the threads must execute the LCD in iteration order.

```

PROGRAM ORDERED ! DEMONSTRATES ORDERED SECTIONS USING CPS
                ! LOW LEVEL SYNCHRONIZATION
REAL X(1000), Y(1000)
INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
INTEGER M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS, M_INIT32, M_FREE32)
INTEGER ORDSEM, SEMCNT
EXTERNAL ORDWORK
PARGS(1) = -2 ! ALLOCATE THREADS CALLING THREAD'S NODE
PARGS(2) = 2  ! MINIMUM OF 2 THREADS
PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM OF NPROCS THREADS
PARGS(4) = 1  ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
SEMCNT = 0
.
. ! SERIAL CODE
.
IERR = M_INIT32(ORDSEM,SEMCNT) ! ALLOCATE ORDSEM
IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
C SPAWN THREADS:
  ITHREAD = CPS_PPCALLN(PARGS,ORDWORK,5,X,Y,NTHR,ORDSEM,SEMCNT)
  IF (ITHREAD .LT. 0) PRINT *, "PPCALLN FAILED"
  IERR = M_FREE32(ORDSEM)
  IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
.
. ! SERIAL CODE
.
END

SUBROUTINE ORDWORK (X, Y, NTHR,ORDSEM,SEMCNT)
REAL X(1000), Y(1000)
INTEGER CPS_NSTHEADS, CPS_STID, M_FETCH32
INTEGER ORDSEM,SEMCNT,STID, NTHR, CNTVAL
INTEGER M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32
C$DIR SYNC_ROUTINE(CPS_NSTHEADS, CPS_STID, M_FETCH32)
C$DIR SYNC_ROUTINE(M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32)
STID = CPS_STID() ! GET MY STID
NTHR = CPS_NSTHEADS() ! GET NUMBER OF THREADS SPAWNED

```

```

C ACTUAL COMPUTATION:
  DO J = 2+STID, 1000, NTHR ! CYCLIC DECOMPOSITION
  .
  . ! DEPENDENCE-FREE PARALLEL CODE
  .
10  CNTVAL = M_FETCH32(ORDSEM) ! GET SEMAPHORE COUNTER VALUE
    IF(CNTVAL .EQ. STID) THEN ! IF IT'S MY STID'S TURN
C PERFORM LCD COMPUTATION:
    X(J) = X(J-1) + Y(J)
    IF(CNTVAL .GE. NTHR-1) THEN ! HIGHEST STID
      IERR = M_FETCH_AND_CLEAR32(ORDSEM) ! RESETS COUNTER
      IF(IERR .LT. 0) PRINT*, "FETCH-CLEAR FAILED"
    ELSE ! ALL OTHER STIDS INCREMENT COUNTER:
      IERR = M_FETCH_AND_INC32(ORDSEM)
      IF(IERR .LT. 0) PRINT*, "FETCH-INC FAILED"
    ENDIF
  ELSE
    GOTO 10 ! LOOP AND TRY AGAIN IF CNTVAL .NE. STID
  ENDIF
ENDDO
RETURN
END

```

This example uses a cyclic decomposition in the parallel  $J$  loop because, by definition, ordered sections must be executed in iteration order, and this is impossible using a block decomposition.

As in the example in the “Cyclic parallelism” section, here the starting index is offset according to spawn thread ID and the loop steps by the number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop;  $NTHR$  elements are computed per step. Contiguous  $STIDs$  compute contiguous elements.

The loop is ordered by the first `IF` statement in the loop, which only allows the body of the loop (including the LCD) to execute if the counter associated with the semaphore `ORDSEM` is equal to the current `STID`. This counter is incremented (or reset when the highest `STID` is reached) in the body of the loop, forcing the threads to execute in iteration order. The counter associated with `ORDSEM` controls access to the LCD; no explicit semaphore lock is needed.

Substantial nonordered work must be present in this loop to make the overhead of the ordered section worthwhile. Assuming this condition is met, once all the threads pass through the ordered section once, their execution of the nonordered code will be staggered such that they will stay busy while they are outside the ordered code.

The ordered parallelism described here is similar to that achieved through use of compiler directives in the "Ordered sections" section of Chapter 6, "Advanced shared memory programming."

# Glossary

---

## A

### **ABI**

Application Binary Interface. A software implementation that allows the same executable programs to run on computer systems with different hardware architectures. Convex SPP Series systems are implemented with an ABI that allows compatibility with other computer systems that run Hewlett-Packard's HP-UX operating system.

### **absolute address**

An address that does not undergo virtual-to-physical address translation when used to reference memory or the I/O register area.

### **accumulator**

A variable used to accumulate value. Accumulators are typically assigned a function of themselves, and this can create dependencies when done in loops.

### **actual argument**

In Fortran, a value that is passed by a call to a procedure (function or subroutine). The actual argument appears in the source of the calling procedure; the argument that appears in the source of the called procedure is a *dummy argument*. C conventions refer to actual arguments as *actual parameters*.

### **actual parameter**

In C, a value that is passed by a call to a procedure (function). The actual parameter appears in the source of the calling procedure; the parameter that appears in the source of the called procedure is a *formal parameter*. Fortran conventions refer to actual parameters as *actual arguments*.

### **address**

A number used by the operating system to identify a storage location.

**address space**

Memory space, either physical or virtual, available to a process.

**Agent**

The gate array on Convex SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *CPU Agent* and the *CPA*.

**alias**

An alternative name for some object, especially an alternative variable name that refers to a memory location. Aliases can cause data dependencies, which prevent the compiler from parallelizing parts of a program.

**alignment**

A condition in which the address, in memory, of a given data item is integrally divisible by a particular integer value, often the size of the data item itself. Alignment simplifies the addressing of such data items.

**ALL**

Acronym for Assembler, Loader, and Libraries.

**allocatable array**

In Fortran 90, a named array whose rank is specified at compile time, but whose bounds are determined at run time.

**allocate**

An action preformed by a program at runtime in which memory is reserved to hold data of a given type. In Fortran 90, this is done through the creation of *allocatable arrays*. In C, it is done through the dynamic creation of memory blocks using `malloc`.

**ALU**

Arithmetic logic unit. A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

**Amdahl's law**

A statement that the ultimate performance of a computer system is limited by the slowest component. In the context of SPP systems this is interpreted to mean that the serial component of the application code will restrict the maximum speed-up that is achievable.

**American National Standards Institute (ANSI)**

A repository and coordinating agency for standards implemented in the U.S. Its activities include the production of Federal Information Processing (FIPS) standards for the Department of

Defense (DoD).

### **ANSI**

See *American National Standards Institute*.

### **APC**

Application Compiler. A Convex compiler that performs *interprocedural optimization* and simplifies the compilation of C and Fortran program modules into a single application. This compiler performs many optimizations automatically that must be performed by hand with other compilers, such as procedure inlining.

### **apparent recurrence**

A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called a *potential recurrence*.

### **Application Binary Interface (ABI)**

A software implementation that allows the same executable programs to run on computer systems with different hardware architectures. Convex SPP Series systems are implemented with an ABI that allows compatibility with other computer systems that run Hewlett-Packard's HP-UX operating system.

### **Application Compiler (APC)**

A Convex compiler that performs *interprocedural optimization* and simplifies the compilation of C and Fortran program modules into a single application. This compiler performs many optimizations automatically that must be performed by hand with other compilers, such as procedure inlining.

### **argument**

In Fortran, either a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called (*dummy argument*) or the variable or constant that is passed by a call to a procedure (*actual argument*). C conventions refer to arguments as *parameters*.

### **arithmetic logic unit (ALU)**

A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

### **array**

An ordered structure of operands of the same data type. The structure of an array is defined by its rank, shape, and data type.

### **array section**

A Fortran 90 construct that defines a subset of an array by providing starting and ending elements and strides for each

dimension. For an array  $A(4, 4)$ ,  $A(2:4:2, 2:4:2)$  is an array section containing only the evenly indexed elements  $A(2, 2)$ ,  $A(4, 2)$ ,  $A(2, 4)$ , and  $A(4, 4)$ .

**array-valued argument**

In Fortran 90, an *array section* that is an actual argument to a subprogram.

**ASCII**

American Standard Code for Information Interchange. This encodes printable and non-printable characters into a range of integers.

**assembler**

A program that converts assembly language programs into executable machine code.

**assembly language**

A programming language whose executable statements can each be translated directly into a corresponding machine instruction of a particular computer system.

**automatic array**

In Fortran, an array of explicit rank that is not a dummy argument and is declared in a subprogram.

**B**

**balancing**

Expressions are represented internally as trees whose height corresponds to the depth of the expression. Tree-height reduction or balancing is an optimization that attempts to simplify expressions by shortening their tree height.

**bandwidth**

A measure of the rate at which data can be moved through a device or circuit. Bandwidth is usually measured in millions of bytes per second (Mbytes/sec) or millions of bits per second (Mbits/sec).

**bank conflict**

An attempt to access a particular memory bank before a previous access to the bank is complete.

**barrier**

A structure used by the compiler in barrier synchronization. Also sometimes used to refer to the construct used to implement barrier synchronization. See also *barrier synchronization*.

**barrier synchronization**

A control mechanism used in parallel programming that ensures

all threads have completed an operation before continuing with the next operation. On SPP Series machines, barrier synchronization can be automated by certain CPSlib routines and compiler directives. See also *barrier*.

**basic block**

A linear sequence of machine instructions with a single entry and a single exit.

**bit**

A binary digit.

**block-shared memory**

Memory that is addressed by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the subcomplex-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a `block_shared` array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division. Compare with *node-private memory*, *thread-private memory*, *near-shared memory*, and *far-shared memory*.

**blocking factor**

Integer representing the stride of the outer strip of a pair of loops created by blocking.

**branch**

A class of instructions which change the value of the program counter to a value other than that of the next sequential instruction.

**byte**

A group of contiguous bits starting on an addressable boundary. In Convex machines, a byte is 8 bits in length.

---

**C**

**cache**

A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the main memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory, and can be accessed with substantially less latency. Convex computers employ separate data and instruction cache memories.

**cache, direct mapped**

A form of cache memory which addresses encached data by a function of the data's virtual address. On SPP1000 Series computers, the cache address is identical to the least significant 20 bits of the data's virtual address. This means cache thrashing can occur when the virtual addresses of two data items are an exact multiple of 1 Mbyte (20 bits) apart. On SPP1200 Series computers, the main cache address is identical to the least significant 18 bits of the data's virtual address. Cache thrashing can therefore occur on SPP1200 machines when the virtual addresses of two data items are an exact multiple of 256 kbytes (18 bits) apart.

**cache, fully associative**

A form of cache memory which attempts to prevent encached data from being overwritten by storing an incoming element in a cache location that is determined using a hashing algorithm. The incoming element's virtual address is irrelevant. Unlike a direct mapped cache, a fully associative cache will never displace a cache line unless the cache is full. SPP1200 Series machines use 2 kbyte on-chip fully associative assist caches to mitigate cache thrashing.

**cache hit**

A *cache hit* occurs if data to be loaded is residing in the cache.

**cache line**

A chunk of contiguous data that is copied into a cache in one operation. On SPP Series computers, processor cache lines consist of 32 bytes of data; when a processor cache miss occurs and data must be fetched from outside the processor cache, the requested data is brought in as part of a 32 byte cache line. CTIcache lines consist of 64 bytes of data (two contiguous processor cache lines); when a CTIcache miss occurs, the requested data is brought into the CTIcache as part of a 64 byte cache line.

**cache memory**

A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the main memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory, and can be accessed with substantially less latency. Convex computers employ separate data and instruction cache memories.

**cache miss**

A *cache miss* occurs if data to be loaded is not residing in the cache.

**cache purge**

The act of invalidating or removing entries in a cache memory.

**cache thrashing**

*Cache thrashing* occurs when two or more data items that are frequently needed by the program map to the same cache address. In this case, each time one of the items is encached it overwrites another needed item, causing constant cache misses and impairing data reuse.

**CCMC**

The Convex Coherent Memory Controller gate array, which, among other tasks, provides an interface to the CTI interface and to memory and maintains cache coherency information. Each pair of processors has a CCMC.

**central processing unit (CPU)**

The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

**clock cycle**

The duration of the square wave pulse sent throughout a computer system to synchronize operations. The clock cycle time for SPP1000 Series systems is  $\approx 10$  nanoseconds; for SPP1200 Series systems, it is  $\approx 8.33$  nanoseconds.

**clone**

A compiler-generated copy of a loop or procedure. When the SPP Series procedural compilers generate code for a parallelizable loop, they generate two versions: a serial clone and a parallel clone. See also *dynamic selection*.

**code**

A computer program, either in source form or in the form of an executable image on a machine.

**coherency**

A term frequently applied to caches. If a data item is referenced by a particular processor on a multi-processor system, the data is copied into that processor's cache and is updated there if the processor modifies the data. If another processor references the data while a copy is still in the first processor's cache, a mechanism is needed to ensure that the second processor does not use an outdated copy of the data from memory. The state that is achieved when both processors' caches always have the latest value for the data is called cache coherency. On SPP Series computers, an item of data may reside concurrently in several processors' caches.

**column-major order**

Memory representation of an array such that the columns are stored contiguously. For example, given a two-dimensional array  $A(3, 4)$ , the array element  $A(3, 1)$  immediately precedes

element  $A(1, 2)$  in memory. This is the default storage method for arrays in Fortran.

**compiler**

A computer program that translates computer code written in a high-level programming language, such as Fortran, into equivalent machine language.

**Compiler Parallel Support library (CPSlib)**

A library of low-level parallelization and synchronization routines. Refer to Appendix D, "Compiler Parallel Support library," for more information.

**complex**

The complete set of processor and memory resources available on a Convex SPP Series system.

**concurrent**

In parallel processing, threads that can execute at the same time are called concurrent threads.

**conditional induction variable**

A loop induction variable that is not incremented on every iteration.

**constant folding**

Replacement of an operation on constant operands with the result of the operation.

**constant propagation**

The automatic compile-time replacement of variable references with a constant value previously assigned to that variable. Constant propagation is performed within a single procedure by conventional compilers, and between procedures by the Application Compiler.

**control parallel programming**

A type of parallel programming in which different functional sections of code are assigned to different processors for simultaneous execution. See also *data parallel programming*.

**conventional compiler**

A compiler that cannot perform interprocedural optimization. This term encompasses all Convex compilers except the *Application Compiler* and virtually all compilers available from other vendors.

**counter**

A variable that is used to count the number of times an operation occurs. CPSlib semaphores are associated with counters so that

they can facilitate barrier synchronization or the creation of ordered sections.

### **CPA**

**CPU Agent.** The gate array on Convex SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *CPU Agent* and the *CPA*.

### **CPSlib (Compiler Parallel Support library)**

A library of low-level parallelization and synchronization routines. Refer to Appendix D, "Compiler Parallel Support library," for more information.

### **CPU**

**Central processing unit.** The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

### **CPU Agent**

The gate array on Convex SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *Agent* and the *CPA*.

### **CPU-private memory**

Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address which maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory*, *near-shared memory*, *far-shared memory*, and *block-shared memory*.

### **CPU time**

The amount of time the CPU requires to execute a program. Because programs share access to a CPU, the wall clock time of a program may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall clock time. (See *wall clock time*.)

### **critical section**

A portion of a parallel program that can be executed by only one thread at a time.

### **crossbar**

A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode of a Convex SPP Series system. Because the crossbar is nonblocking, all ports can run at

full bandwidth simultaneously provided there isn't contention for a particular port.

### **CSR**

Control/Status Register. A CSR is a software-addressable hardware register used to hold control information or state.

### **CTIcache**

A partition of physical memory that exists on each hypernode and is used to store copies of global data fetched from other hypernodes.

### **CTI interface**

The hardware interface between the CTI rings and the CCMC.

### **CTI (Coherent Toroidal Interface) ring**

The ring interconnect that connects all the hypernodes of a multi-hypernode Convex SPP Series system together in a ring topology. While the CTI ring is derived from the IEEE SCI standard, complete compatibility is sacrificed to provide lower latencies.

---

## **D**

### **data cache (Dcache)**

A small cache memory with a one clock cycle access time. This cache holds prefetched and current data. On SPP1000 Series systems, it is 1 Mbyte in size. SPP1200 Series systems have a 2 kbyte on-chip cache and a 256 kbyte off-chip cache. See also *cache, fully associative* and *cache, direct mapped*.

### **data dependency**

A relationship between two statements in a program, such that one statement must precede the other to produce the intended result. (See also *loop carried dependence (LCD)* and *loop independent dependence (LID)*.)

### **data localization**

Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

### **data parallel programming**

A type of parallel programming in which the data upon which an operation is to be performed is divided among several threads, which execute concurrently in separate processors.

### **data type**

A property of a data item which determines how its bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in

the operand. Some example data types include INTEGER, int, REAL, and float.

**Dcache**

Data cache. On Convex SPP Series systems, a 1-Mbyte cache memory with a one clock cycle access time. This cache holds prefetched and current data.

**deadlock**

A condition in which a thread waits indefinitely for some condition or action that can not or will not occur.

**direct memory access (DMA)**

A method for gaining direct access to main memory and achieving data transfers without involving the Central Processing Unit.

**distributed memory**

A memory architecture used in multi-CPU systems, in which the system's memory is physically divided among the processors. In most distributed memory architectures, distributed memory is accessible from only a single processor; sharing of data requires explicit message passing. Convex's GSM is an exception. See also *global memory, globally shared memory (GSM)*.

**distributed part**

A loop generated by the compiler in the process of loop distribution.

**DMA**

Direct memory access. A method for gaining direct access to main memory and achieving data transfers without involving the Central Processing Unit.

**double**

A double-precision floating-point number that is stored in 64 bits in C.

**doubleword**

A primitive data operand which is 8 bytes (64 bits) in length. Also called a *longword*. See also *word*.

**dummy argument**

In Fortran, a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called. The dummy argument appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual argument*. C conventions refer to dummy arguments as *formal parameters*.

**dynamic selection**

The process by which the compiler chooses the appropriate runtime clone of a loop. See also *clone*.

---

**E****encache**

To copy data into a cache.

**exception**

A hardware-detected event that interrupts the running of a program, process, or system. See also *fault*.

**execution stream**

A series of instructions executed by a CPU.

---

**F****far-shared memory**

Memory that is addressed by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. Far-shared memory is physically distributed by pages round-robin to all the hypernodes in the subcomplex, so the virtual address maps to a single physical address located on one of the hypernodes. Access latencies therefore vary as a function of hypernode and data element. Compare with *node-private memory*, *thread-private memory*, *near-shared memory*, and *block-shared memory*.

**fault**

A type of *interruption* caused by an instruction which requests a legitimate action which cannot be carried out immediately due to a system problem.

**floating point**

A numerical representation of a real number. On Convex SPP Series computers, a floating point operand has a sign (positive or negative) part, an exponent part, and a fraction part. The fraction is a fractional representation. The exponent is the value used to produce a power of two scale factor (or portion) that is subsequently used to multiply the fraction to produce an unsigned value.

**FLOPS**

Floating-point operations per second. A standard measure of computer processing power in the scientific community.

**formal parameter**

In C, a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called. The formal parameter appears in the source of the called procedure;

---

the parameter that appears in the source of the calling procedure is an *actual parameter*. Fortran conventions refer to formal parameters as *dummy arguments*.

### **Fortran**

A high-level software language used mainly for scientific applications.

### **Fortran 90**

The international standard for Fortran adopted in 1991.

### **function**

A procedure whose call can be imbedded within another statement, such as an assignment or test. Any procedure in C or a procedure defined as a `FUNCTION` in Fortran.

### **functional unit (FU)**

A part of a *CPU* that performs a set of operations on quantities stored in *registers*.

---

## **G**

### **gate**

A construct that restricts execution of a block of code to a single thread. A thread locks a gate on entering the gated block of code and unlocks the gate on exiting the block. When the gate is locked, no other threads can enter. Compiler directives can be used to automate gate constructs; gates can also be implemented using *semaphores*.

### **Gbyte**

See *gigabyte*.

### **gigabyte**

1073741824 ( $2^{30}$ )bytes.

### **globally shared memory (GSM)**

A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as shared virtual memory or global virtual memory.

### **global optimization**

A restructuring of program statements that is not confined to a single basic block. Global optimization, unlike interprocedural optimization, is confined to a single procedure. Global optimization is done by all Convex compilers at optimization level `-O1` and above.

**global memory**

Memory that is accessible from several processors. See also *distributed memory*, *globally shared memory (GSM)*.

**global variable**

A variable whose scope is greater than a single procedure. In C programs, a global variable is a variable that is defined outside of any one procedure. Fortran has no global variables per se, but COMMON blocks can be used to make certain memory locations globally accessible.

**global virtual memory**

Global shared memory. A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as shared virtual memory or global virtual memory.

**granularity**

In the context of parallelism, a measure of the relative size of the computation done by a thread or parallel construct. Performance is generally an increasing function of the granularity. In higher-level language programs, possible sizes are routine, loop, block, statement, and expression. Fine granularity is exhibited by parallel loops, tasks and expressions; coarse granularity is exhibited by parallel processes.

---

**H****hand-rolled loop**

A loop, more common in Fortran than C, which is constructed using IF tests and GOTO statements rather than a language-provided loop structure such as DO.

**hidden alias**

An alias that, because of the structure of a program or the standards of the language, goes undetected by the compiler. Hidden aliases can result in undetected *data dependencies*, which may result in wrong answers.

**High Performance Fortran (HPF)**

An ad-hoc language extension of Fortran 90 that provides user-directed data distribution and alignment. HPF is not a standard, but rather a set of features desirable for parallel programming.

**hoist**

An optimization process that moves a memory load operation

from within a loop to the basic block preceding the loop.

**host**

In the context of PVM, A computer system into which communications hardware and software have been installed, and which is accessed as a separate entity by other hosts.

**HP**

Hewlett-Packard, the manufacturer of the PA-RISC chips used as CPUs in Convex SPP Series systems.

**HP-UX**

Hewlett-Packard's UNIX-based operating system for its PA-RISC workstations.

**hypercube**

A topology used in some massively parallel processing systems. Each processor is connected to its binary neighbors. The number of processors in the system is always a power of two; that power is referred to as the dimension of the hypercube. For example, a 10-dimensional hypercube has  $2^{10}$ , or 1,024 processors.

**hypernode**

In Convex SPP Series systems, a set of up to eight processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system microkernel. An SPP system consists of one or more hypernodes, with a high speed *CTI ring* connecting the hypernodes. When discussing multi-dimensional parallelism or memory classes, hypernodes are generally called nodes.

---

**lcache**

Instruction cache. On SPP1000 systems, a 1-Mbyte cache memory with a one clock cycle access time. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On SPP1200 systems it is 256 kbytes in size.

**IEEE**

Institute for Electrical and Electronic Engineers. An international professional organization and a member of ANSI and ISO.

**inlining**

The replacement of a procedure (function or subroutine) call, within the source of a calling procedure, by a copy of the called procedure's code.

**Institute for Electrical and Electronic Engineers (IEEE)**

An international professional organization and a member of ANSI

and ISO.

**instruction**

One of the basic operations performed by a CPU.

**instruction cache (lcache)**

On SPP1000 systems, a 1-Mbyte cache memory with a one clock cycle access time. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On SPP1200 systems it is 256 kbytes in size.

**instruction mnemonic**

A symbolic name for a machine instruction.

**integral division**

Division that results in a whole number solution with no remainder. For example, 10 is integrally divisible by 2, but not by 3.

**interface**

A logical path between any two modules or systems.

**interleaved memory**

Memory that is divided into multiple banks to permit concurrent memory accesses. The number of separate memory banks is referred to as the memory stride.

**interprocedural optimization**

Automatic analysis of relationships and interfaces between all subroutines and data structures within a program. Traditional compilers analyze only the relationships within the procedure being compiled. The *Convex Application Compiler* is capable of performing interprocedural optimization.

**interprocessor communication**

The process of moving or sharing data, and synchronizing operations between processors on a multiprocessor system.

**intrinsic**

A function or subroutine that is an inherent part of a computer language. For example, `SQRT` is a Fortran intrinsic.

---

**J**

**job scheduler**

That portion of the operating system that schedules and manages the execution of all processes.

**join**

The synchronized termination of parallel execution by spawned

tasks or threads.

**jump**

Departure from normal one-step incrementing of the program counter.

---

**K**

**kbyte**

See *kilobyte*.

**kernel**

The core of the operating system where basic system facilities, such as file access and memory management functions, are performed.

**kernel thread identifier (ktid)**

A unique integer identifier (not necessarily sequential) assigned when a thread is created.

**kilobyte**

1024 ( $2^{10}$ ) bytes.

---

**L**

**large application**

An application requiring a virtual address space larger than 4 Gbytes.

**latency**

The time delay between the issuing of an instruction and the completion of the operation. A common benchmark used for comparing SPP systems is the latency of coherent memory access instructions. This particular latency measurement is believed to be a good indication of the *scalability* of an SPP system; low latency equates to low system overhead as system size increases.

**libpvm**

The C programming library (libpvm3.a), Fortran programming library (libfpvm3.a), or group library (libgpvm3.a), supporting the PVM message passing programming style.

**linker**

A software tool that combines separate object code modules into a single object code module or executable program.

**load**

An instruction used to move the contents of a memory location into a register.

**locality of reference**

An attribute of a memory reference pattern that refers to the

---

likelihood of an address of a memory reference being physically close, in terms of the number of clock cycles necessary to access it, to the CPU making the reference.

### **local optimization**

Restructuring of program statements within the scope of a basic block. Local optimization is done by all Convex compilers at optimization level -00 and above.

### **localization**

Data localization. Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

### **logical address**

Logical address space is that address as seen by the application program.

### **logical memory**

Virtual memory. The memory space as seen by the program, which is typically larger than the available physical memory. The virtual memory of a Convex computer can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). SPP-UX maps this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called logical memory.

### **longword (l)**

Doubleword. A primitive data operand which is 8 bytes (64 bits) in length. Also called a *longword*. See also *word*.

### **loop blocking**

A loop transformation that stripmines and interchanges a loop to provide optimal reuse of the encachable loop data.

### **loop carried dependence (LCD)**

A dependence between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependence from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores into an address that is referred to on a different iteration of the loop. To parallelize a loop containing an LCD, you generally must manually synchronize the LCD assignment and manually parallelize the loop.

### **loop constant**

A constant or expression whose value does not change within a loop.

**loop distribution**

The restructuring of a loop nest to create simple loop nests. Loop distribution creates two or more loops, called distributed parts, which can serve to make parallelization more efficient by increasing the opportunities for loop interchange and isolating code that must run serially from parallelizable code. It can also improve data localization and other optimizations.

**loop independent dependence (LID)**

A dependence between two operations executed on the same iteration of all enclosing loops such that one operation must precede the other to produce correct results.

**loop induction variable**

A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount on every iteration. For example, in the following Fortran loop, I, J and K are induction variables, but L is not.

```
DO I = 1, N
  J = J + 2
  K = K + N
  L = L + I
ENDDO
```

**loop interchange**

The reordering of nested loops. Loop interchange is generally done to increase the granularity of the parallelizable loop(s) present or to allow more efficient access to loop data.

**loop invariant**

Loop constant. A constant or expression whose value does not change within a loop.

**loop invariant computation**

An operation that yields the same result on every iteration of a loop.

**loop replication**

The process of transforming one loop into more than one loop to facilitate an optimization. The optimizations that replicate loops are IF-DO and if-for optimizations, dynamic selection, loop unrolling, unroll and jam, and loop blocking.

---

**M****machine exception**

A fatal error in the system that cannot be handled by the operating system. See also *exception*.

**main memory**

On SPP Series systems, physical memory other than the processor caches that is not allocated as part of the CTIcache.

**main procedure**

A procedure invoked by the operating system when an application program starts up. The main procedure is the main program in Fortran; in C, it is the function `main()`.

**main program**

In a Fortran program, the program section invoked by the operating system when the program starts up.

**Massively Parallel Processor (MPP)**

A computer architecture in which several to thousands of processors are combined to work simultaneously on solving complex problems.

**Mbyte**

See *megabyte (Mbyte)*.

**megabyte (Mbyte)**

1048576 ( $2^{20}$ ) bytes.

**megaflops (MFLOPS)**

One million floating-point operations per second.

**memory bank conflict**

An attempt to access a particular memory bank before a previous access to the bank is complete.

**memory management**

The hardware and software that control memory page mapping and memory protection.

**message**

Data copied from one process to another (or the same) process. The copy is initiated by the sending process, which specifies the receiving process. The sending and receiving processes need not share a common address space. (Note: depending on the context, a process may be a *thread*.)

**message passing**

A type of programming in which program modules (often running on different processors or different hosts) communicate with each other by means of system library calls that package, transmit, and receive data. All message passing library calls must be explicitly coded by the programmer.

**MIMD (multiple instruction stream multiple data stream)**

A computer architecture that uses multiple processors, each processing its own set of instructions simultaneously and independently of others. Compare with *SIMD*.

**MPP**

Massively Parallel Processor. A computer architecture in which several to thousands of processors are combined to work simultaneously on solving complex problems.

**multiprocessing**

The creation and scheduling of processes on any subset of CPUs in a system configuration.

**mutex**

A variable used to construct an area of *mutual exclusion*. When a mutex is locked, entry to the area is prohibited; when the mutex is free, entry is allowed.

**mutual exclusion**

A protocol that prevents access to a given resource by more than one thread at a time.

---

**N****near-shared memory**

Memory that is addressable by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. Near-shared memory resides physically on the hypernode from which it was allocated, and is accessed with lowest latency from that hypernode. Access latencies are higher from other hypernodes. Compare with *thread-private memory*, *node-private memory*, *block-shared memory*, and *far-shared memory*.

**negate**

An instruction that changes the sign of a number.

**network**

A system of interconnected computers that enables machines and their users to exchange information and share resources. Convex SPP Series systems provide support for *FDDI* networks.

**node**

On SPP Series computers, a node is equivalent to a *hypernode*. The term "node" is generally used in place of hypernode when discussing parallelism or memory classes.

**node-private memory**

Memory residing on a hypernode that is accessible only by CPUs on the same hypernode in a Convex SPP Series system. A node-private data object has a unique virtual address by which all

threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode. Compare with *thread-private memory*, *near-shared memory*, *block-shared memory*, and *far-shared memory*.

### **non uniform memory access (NUMA)**

This term describes memory access times in systems such as the SPP Series, in which different types of memory result in non-uniform access times.

### **nonblocking crossbar**

A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode of a Convex SPP Series system. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously provided there isn't contention for a particular port.

### **NUMA**

Non uniform memory access. This term describes memory access times in systems such as the SPP Series, in which different types of memory result in non-uniform access times.



### **offset**

In the context of a process address space, an integer value that is added to a base address to calculate a memory address. Offsets in Convex SPP Series systems are 32-bit values, and must keep address values within a single 4 Gbyte memory space.

### **opcode**

A predefined sequence of bits in an instruction that specifies the operation to be performed.

### **operating system**

The program that manages the resources of a computer system. The Convex SPP Series system uses the SPP-UX operating system. SPP-UX is compatible with Hewlett-Packard's HP-UX operating system.

### **optimization**

The refining of application software programs to minimize processing time. Optimization takes maximum advantage of a computer's hardware features and minimizes input/output traffic and idle processor time.

### **optimization level**

The degree to which source code is optimized by the compiler. The Convex Fortran and C compilers have five levels of

optimization, level -no, -00, -01, -02, and -03.

**oversubscript**

An array reference that falls outside declared bounds.

**oversubscription**

In the context of parallel threads, a process attribute that permits the creation of more threads within a process than the number of processors available to the process.

---

**P**

**PA-RISC**

The Hewlett-Packard Precision Architecture reduced instruction set processor chip. This is the processor chip used in Convex SPP Series systems.

**packet**

A group of related items. A packet may refer to the arguments of a subroutine or to a group of bytes that is transmitted over a network.

**page**

A page is the unit of virtual or physical memory controlled by the memory management hardware and software. On Convex SPP Series systems, a page is 4 K (4,096) contiguous bytes. See also *virtual memory*.

**page fault**

A page fault occurs when a process requests data that is not currently in main memory. This requires the operating system to retrieve the page containing the requested data from disk.

**page frame**

A page frame is the unit of physical (main) memory in which pages are placed. Referenced and modified bits associated with each page frame aid in memory management.

**parallel optimization**

The transformation of source code into parallel code (parallelization) and restructuring of code to enhance parallel performance.

**Parallel Virtual Machine (PVM)**

A message-passing and process control library available on Convex SPP Series systems.

**parallelization**

The process of transforming serial code to a form of code that can run simultaneously on multiple CPUs while preserving semantics. At optimization level -03, the Convex compilers

automatically parallelize loops in your program and recognize compiler directives with which you can manually specify parallelization of both loops and tasks.

### **parallelization, loop**

The process of splitting a loop into several smaller loops, each of which operates on a subset of the data of the original loop, and generating code to run these loops on separate processors in parallel.

### **parallelization, ordered**

The process of splitting a loop into several smaller loops, each of which iterates over a subset of the original data with a stride equal to the number of loops created, and generating code to run these loops on separate processors. Each iteration in an ordered parallel loop begins execution in the original iteration order, allowing dependencies within the loop to be synchronized to yield correct results via gate constructs.

### **parallelization, stride-based**

The process of splitting up a loop into several smaller loops, each of which iterates over several discontinuous chunks of data, and generating code to run these loops on separate processors in parallel. Stride-based parallelism can only be achieved manually by using compiler directives or CPSlib functions.

### **parallelization, strip-based**

The process of splitting up a loop into several smaller loops, each of which iterates over a single contiguous subset of the data of the original loop, and generating code to run these loops on separate processors in parallel. Strip-based parallelism is the default for automatic parallelism and for directive-initiated loop parallelism in absence of the `chunk_size = n` or `ordered` attributes.

### **parallelization, task**

The process of splitting up source code into independent sections which can safely be run in parallel on available processors. Convex programming languages provide compiler directives and pragmas which allow you to identify parallel tasks in source code.

### **parameter**

In C, either a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called (*formal parameter*) or the variable or constant that is passed by a call to a procedure (*actual parameter*). In Fortran, a symbolic name for a constant.

### **path**

An environment variable that you set in your shell configuration

file that allows you to access commands in various directories without having to specify a complete path name.

**physical address**

A unique identifier that selects a particular location in the computer's memory. Because SPP-UX supports virtual memory, programs running on SPP Series computers address data by its virtual address; SPP-UX then maps this address to the appropriate physical address. See also *virtual address*.

**physical address space**

The set of possible addresses for a particular physical memory.

**physical memory**

Computer hardware that stores data. SPP Series systems can contain up to 2 Gbytes of physical memory per hypernode, for a total of 32 Gbytes of physical memory on a full 16-hypernode system.

**pipeline**

An overlapping operating cycle function that is used to increase the speed of computers. Pipelining provides a means by which multiple operations occur concurrently by beginning one instruction sequence before another has completed. Maximum efficiency is achieved when the pipeline is "full", i.e. when all stages are operating on separate instructions.

**pipelining**

Issuing instructions in an order that best utilizes the pipeline.

**pointer tracking**

A powerful alias-detecting algorithm used by the Application Compiler, which tracks the memory locations to which each pointer can be set and enables concurrent execution of code which does not have overlapping memory references.

**procedure**

A unit of program code. In Fortran, a function, subroutine or main program; in C, a function.

**process**

A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.

**process memory**

The portion of system memory that is used by an executing process.

**programming model**

A description of the features available to efficiently program a certain computer architecture.

**program unit**

A procedure or main section of a program.

**PVM**

Parallel virtual machine. A message-passing and process control library available on Convex SPP Series systems.

---

**Q****queue**

A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in, first-out (FIFO).

---

**R****rank**

The number of dimensions of an array.

**read**

A memory operation in which the contents of a memory location are copied and passed to another part of the system.

**recurrence**

A cycle of dependencies among the operations within a loop in which an operation in one iteration depends on the result of a following operation that executes in a previous iteration.

**recursion**

An operation that is defined at least in part by a repeated application of itself.

**recursive call**

A condition in which the sequence of instructions in a procedure causes the procedure itself to be invoked again. Such a procedure must be compiled for *reentrancy*.

**reduced instruction set computer (RISC)**

An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in SPP Series computers employs a RISC architecture.

**reduction**

An arithmetic operation that performs a transformation on an array to produce a scalar result.

**reenfrancy**

The ability of a program unit to have multiple versions in existence that may execute in parallel. Each version maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables. Procedures must be compiled for reentrancy in order to be invoked in parallel or to be used for recursive calls. Reentrant compilation is the default on SPP Series computers.

**reference**

Any operation that requires a cache line to be encached; this includes load as well as store operations, since writing to any element in a cache line requires the entire cache line to be encached.

**register**

A hardware entity that contains an address, operand, or instruction status information.

**reuse, data**

In the context of a loop, the ability to use data fetched for one loop operation in another operation. In the context of a cache, reusing data that was encached for a previous operation; since data is fetched as part of a cache line, if any of the other items in the cache line are used before the line is flushed to memory, reuse has occurred.

**reuse, spatial**

Reusing data that resides in the cache as a result of the fetching of another piece of data from memory. Typically, this involves using array elements that are contiguous to (and therefore part of the cache line of) an element that has already been used, and therefore is already encached.

**reuse, temporal**

Reusing a data item that has been used previously.

**RISC**

Reduced instruction set computer. An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in SPP Series computers employs a RISC architecture.

**rounding**

A method of obtaining a representation of a number that has less precision than the original in which the closest number representable under the lower precision system is used.

**row-major order**

Memory representation of an array such that the rows of an array are stored contiguously. For example, given a two dimensional array `A[3][4]`, array element `A[0][3]` immediately precedes `A[1][0]` in memory. This is the default storage method for arrays in C. It can be selected in Convex Fortran by using the `ROWWISE` compiler directive.

---

**S****Scalable Parallel Processor (SPP)**

A computer architecture that permits a sufficiently parallel application to run with a relatively small number of processors or in a processor array containing hundreds to thousands of processors. A key design goal for SPP systems is to enable performance to increase linearly with respect to its number of processors.

**SCI**

Scalable Coherent Interface. This is defined by IEEE standard 1596-1992. The interface is physically defined as a pair of 18-bit, differential ECL, unidirectional links which are clocked at 250 Mhz. Each link provides 16 bits of data with two control signals. Data is sampled on both the rising and falling edges of the clock. This interface provides the basis for the CTI rings used in Convex SPP Series systems; however, total compatibility with the standard has been sacrificed to provide increased performance.

**scope**

The domain in which a variable is visible in source code. The rules that determine scope are different for Fortran and C.

**semaphore**

An integer variable assigned one of two values: one value to indicate that it is "locked", and another to indicate that it is "free". Semaphores can be used to synchronize parallel threads. CPSlib provides a set of manipulation functions to facilitate this.

**shape**

The number of elements in each dimension of an array.

**shared virtual memory**

Globally shared memory. A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as shared virtual memory or global virtual memory.

**shell**

An interactive command interpreter that is the interface between the user and the operating system.

**SIMD (single instruction stream multiple data stream)**

A computer architecture that performs one operation on multiple sets of data. One processor is used for the control logic and the remaining processors are used as slaves. Compare with *MIMD* (*multiple instruction stream multiple data stream*).

**single**

A single-precision floating-point number stored in 32 bits. See also *double*.

**SMP**

Symmetric multiprocessor. A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no master or slave processors; the operating system runs on any or all of the processors.

**socket**

An endpoint used for interprocess communication.

**socket pair**

Bidirectional pipes that enable application programs to set up two-way communication between processes that share a common ancestor.

**source code**

The uncompiled version of a program, written in a high-level language such as C or Fortran.

**source file**

A file that contains program source code.

**space**

A contiguous range of virtual addresses within the system-wide virtual address space. Spaces are 4 Gbytes in size in Convex SPP Series systems.

**spatial reference**

An attribute of a memory reference pattern that pertains to the likelihood of a subsequent memory reference address being numerically close to a previously referenced address.

**spawn**

To activate existing threads.

**spawn context**

A parallel loop or task list that initiates the spawning of threads and defines the structure within which the threads' spawn thread

IDs are valid.

### **spawn thread identifier (stid)**

A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

### **SPMD**

Single program multiple data. A single program executing simultaneously on several processors. This is usually taken to mean that there is redundant execution of sequential scalar code on all processors.

### **SPP**

Scalable parallel processor. A computer architecture that permits a sufficiently parallel application to run with a relatively small number of processors or in a processor array containing hundreds to thousands of processors. A key design goal for SPP systems is to enable performance to increase linearly with respect to its number of processors.

### **stack**

A data structure in which the last item entered is the first to be removed. Also referred to as last-in, first-out (LIFO). SPP UX provides every thread with a stack which is used to pass arguments to functions and subroutines and for local variable storage.

### **store**

An instruction used to move the contents of a register to memory.

### **strip length, parallel**

In strip-based parallelism, the amount by which the induction variable of a parallel inner loop is advanced on each iteration of the (conceptual) controlling outer loop.

### **strip mining**

The transformation of a single loop into two nested loops. Conceptually, this is how parallel loops are created by default. A conceptual outer loop advances the initial value of the inner loop's induction variable by the parallel strip length. The parallel strip length is based on the trip count of the loop and the amount of code in the loop body. Strip mining is also used by the data localization optimization.

**subcomplex**

In Convex SPP Series systems, a logical entity that provides control over the allocation of processors and physical memory to different applications and users.

**subroutine**

A software module that can be invoked from anywhere in a program.

**superscalar**

A class of RISC processors that allow multiple instructions to be issued on each clock period.

**Symmetric Multiprocessor (SMP)**

A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no master or slave processors; the operating system runs on any or all of the processors.

**synchronization**

A method of coordinating the actions of multiple threads so that operations occur in the right sequence. When manually optimizing code, you can synchronize programs using compiler directives, calls to library routines, or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

**system administrator (sysadmin)**

The system manager.

**system manager**

The person responsible for the management and operation of a computer system. Also called the system administrator and the sysadmin.

**system subcomplex**

In a Convex SPP Series system, a subcomplex that is automatically created at boot time by the operating system to run system processes, including `init` and processes spawned by `init`. The Subcomplex Manager will not allow users to destroy this subcomplex, nor remove the last processor from this subcomplex.

---

**T****term**

A constant or symbolic name that is part of an *expression*.

**thread**

An independent execution stream that is executed by a CPU. One or more threads, each of which can execute on a different CPU,

make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by Convex compilers, inserted by adding compiler directives to source code, or coded explicitly using library calls or assembly-language.

**thread create**

To activate existing threads.

**thread identifier**

An integer identifier associated with a particular thread. See *thread identifier, kernel (ktid)* and *thread identifier, spawn (stid)*.

**thread identifier, kernel (ktid)**

A unique integer identifier (not necessarily sequential) assigned when a thread is created.

**thread identifier, spawn (stid)**

A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

**thread-private memory**

Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address which maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory, near-shared memory, far-shared memory, and block-shared memory*.

**tree-height reduction**

Expressions are represented internally as trees whose height corresponds to the depth of the expression. Tree-height reduction or balancing is an optimization that attempts to simplify expressions by shortening their tree height.

**trip count**

The number of iterations a loop executes.

---

## U

### **unsigned**

A value that is always positive.

### **user interface**

The portion of a computer program that processes input entered by a human and provides output for human users.

### **utility**

A software tool designed to perform a frequently used support function.

---

## V

### **vector**

An ordered list of items in a computer's memory, contained within an array. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as offsets to the base.

### **vector processor**

A processor whose instruction set includes instructions that perform operations on a *vector* of data (such as a row or column of an array) in an optimized fashion. Convex C Series systems employ vector processors; Convex SPP Series systems do not.

### **virtual address**

The address by which programs access their data. SPP-UX maps this address to the appropriate physical memory address. See also *space*.

### **virtual aliases**

Two different virtual addresses that map to the same physical memory address.

### **virtual machine**

A collection of computing resources configured so that a user or process can access any of the resources, regardless of their physical location or operating system, from a single interface.

### **virtual memory**

The memory space as seen by the program, which is typically larger than the available physical memory. The virtual memory of a Convex computer can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). SPP-UX maps this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called logical memory.

---

## W

### **wall clock time**

The chronological time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m. the following morning, its wall clock time is sixteen hours. Compare with *CPU time*.

### **word**

A contiguous group of bytes that make up a primitive data operand and start on an addressable boundary. In Convex SPP1000 and SPP1200 Series computer systems, a word is four bytes (32 bits) in length. See also *doubleword*.

### **workstation**

A stand-alone computer that has its own processor, memory, and possibly a disk drive and can typically sit on a user's desk.

### **write**

A memory operation in which a memory location is updated with new data.

---

## Z

### **zero**

In floating point number representations, zero is represented by the sign bit with a value of zero and the exponent with a value of zero.

---

# Index

---

## Symbols

- \* (asterisk) entry  
in optimization report 287

---

## A

- ABI
    - defined 341
  - aborts
    - program 237, 251
  - absolute address
    - defined 341
  - accumulator
    - defined 341
  - accumulator variables
    - and floating-point imprecision 238
  - actual argument
    - defined 341
  - actual parameter
    - defined 341
  - address
    - defined 341
  - address space
    - defined 342
  - affinity
    - threads to CPUs 101
  - agent
    - defined 342
  - algebraic simplification 44
  - alias 91
    - defined 342
  - alias addr option 282
  - alias array\_args option 280
  - alias cautious 233
  - alias cautious option 281
  - alias global option 282
  - alias no\_addr option 282
  - alias no\_global option 282
  - alias ptr\_args option 282
  - alias restrict\_args option 283
  - alias standard 233, 281
  - alias worst 233
  - alias worst option 281
- aliases
  - hidden 230
  - potential 280
- aliasing 230
  - and ANSI C 232
  - and C compatibility modes 231
  - and C pointers 231
  - ANSI C 231
  - ANSI C algorithm 232
  - ANSI C, sometimes unsafe 232
  - command line options 233
  - Fortran example 230
  - global variables 231, 236
  - in C 231
  - local variables 231
  - stop variable 235
  - worst-case 231
- align cseries option 280
- align cti option 26, 280
- align spp option 280
- align\_cti directive and pragma 25, 262
- aligning data 25
  - for cseries 280
  - on CTIcache boundaries 280
- alignment
  - and align\_cti 262
  - data 25, 39
  - defined 342
  - of common blocks 280
  - on natural boundaries 280
- ALL
  - defined 342
- alloc\_barrier function 185
- alloc\_barrier\_8 function 185
- alloc\_gate function 185
- alloc\_gate\_8 function 185
- allocatable array
  - defined 342
- allocate
  - defined 342
- allocation
  - of barriers 185
  - of gates 185
  - of registers 36
- ALU
  - defined 342
- Amdahl's law
  - defined 342
- American National Standards Institute (ANSI)
  - defined 342
- analysis column
  - in analysis table 289
  - in test table 289
- analysis table
  - in optimization report 289
- ANSI
  - defined 342

- ANSI C aliasing 232
- ANSI C aliasing algorithm 232
- ANSI Fortran
  - non-compliant argument passing 57
- APC
  - defined 343
- apparent dependency 241, 248
- apparent LCD 248, 250
- apparent LCDs 106
- apparent recurrence
  - defined 343
- application binary interface
  - defined 343
- Application Compiler
  - defined 343
- architecture
  - memory 14
  - organization 11
- architecture overview 11
- argument
  - actual 341
  - defined 343
- arithmetic logic unit (ALU)
  - defined 343
- array
  - defined 343
- array section
  - defined 343
- array table
  - dependencies 291
  - line number column 291
  - optimization column 291
  - variable name column 291
- arrays
  - dimensions and thrashing 23
  - scalar replacement of elements 85
  - storage order 272
- array-valued argument
  - defined 344
- ASCII
  - defined 344
- assembler
  - defined 344
- assembly language
  - defined 344
- ASSIGN statement 47
- assigned GOTO statements 87
- assignment substitution 41
- assignments
  - elimination of redundant 40, 46
- asymmetric threads
  - compiler parallel support library functions 308
- asterisk (\*) entry
  - in optimization report 287
- asymmetric parallelism 303
  - CPSlib example 330
- automatic array

- defined 344
- automatic parallelization 96
  - disabling 127, 276

---

## B

- backward LCDs 105
- balancing
  - defined 344
  - trees 37
- bandwidth
  - defined 344
- bank conflict
  - defined 344
- barrier
  - defined 344
- barrier synchronization
  - defined 344
- barrier\_t data type 184
- barrier8\_t data type 184
- barriers 184, 262
  - allocating 185
  - allocating with cps\_barrier\_alloc function 317
  - and the compiler parallel support library 317
  - CPSlib example 332
  - deallocation 186
  - freeing with cps\_barrier\_free function 317
  - in C 184
  - in Fortran 184
  - incrementing with cps\_barrier function 318
  - wait\_barrier function 187
- basic block 42
  - defined 345
  - optimizations 45
- BEGIN\_TASKS directive 120
- begin\_tasks directive and pragma 262
- bit
  - defined 345
- block parallelism 112
  - CPSlib example 327
- block\_loop directive and pragma 263
- block\_shared directive 263
- block\_shared memory 148
  - dynamic allocation 169
  - static assignments 155
  - using 210
- block\_shared memory class 263
- blocking
  - block\_loop directive and pragma 71, 263
  - explained 65
  - no\_block\_loop directive and pragma 71, 267
- blocking factor
  - defined 345
  - specifying 263
- blockloop compiler option 72
- blockloop option 278

block-shared memory  
  defined 345  
branch  
  defined 345  
buffering  
  ConvexPVM 217  
bye  
  defined 345

---

## C

c\_cond\_lock function 321  
c\_fetch\_and\_add32 function 322  
c\_fetch\_and\_clear32 function 322  
c\_fetch\_and\_dec32 function 322  
c\_fetch\_and\_inc32 function 321  
c\_fetch\_and\_set32 function 322  
c\_fetch32 function 321  
c\_free32 function 320  
c\_init32 function 320  
c\_lock function 320  
c\_unlock function 321  
cache  
  based semaphores 320  
  data 13  
  defined 345  
  instruction 13  
  interconnect 14  
  preventing thrashing 21  
  thrashing 20  
  thrashing and common blocks 23  
  thrashing example 20  
cache addresses 17  
cache coherency 3  
cache hit  
  defined 346  
cache line  
  defined 346  
cache lines 16  
  CTI 16  
  processor 16  
cache memory  
  defined 346  
cache miss  
  defined 346  
-cache option 276  
cache purge  
  defined 346  
cache size  
  specifying 276  
cache thrashing 20  
  defined 347  
  illustrated 21  
caches 13  
caution  
  on NO\_SIDE\_EFFECTS 49

CCMC 13  
  defined 347  
central processing unit (CPU)  
  defined 347  
chunk\_size  
  attribute to loop\_parallel 112  
chunk-based parallelism 112, 114  
  example 115  
clock cycle  
  defined 347  
clone  
  defined 347  
clones  
  and reentrant compilation 128  
  loop 102  
cloning  
  loop 102  
clustered workstations  
  compilers 4  
  configurability 5  
  interprocess communication 4  
  memory 4  
  peripherals 5  
  vs. SPP 3  
code  
  defined 347  
code motion 52, 257  
  and wrong answers 230  
coherency  
  defined 347  
  in caches 3  
coherent memory controller 13  
Coherent Torodial Interconnect 1  
Col. Num. column  
  in test table 289  
column-major order  
  defined 347  
  example 60  
common blocks  
  and cache thrashing 23  
  padding for C Series 280  
common subexpressions  
  elimination of 42, 50  
compile time  
  and -mrl option 277  
compiler  
  defined 348  
compiler directives 261  
  ALIGN\_CTI 262  
  align\_cti 25  
  BARRIER 262  
  BEGIN\_TASKS 120, 262  
  BLOCK\_LOOP 263  
  BLOCK\_SHARED 263  
  CRITICAL\_SECTION 263  
  DYNSEL 103, 263  
  END\_CRITICAL\_SECTION 264

END\_ORDERED\_SECTION 264  
 END\_TASKS 120, 264  
 FAR\_SHARED 264  
 FAR\_SHARED\_POINTER 265  
 GATE 265  
 LOOP\_PARALLEL 265  
 LOOP\_PRIVATE 266  
 misused 229, 240  
 NEAR\_SHARED 266  
 NEAR\_SHARED\_POINTER 267  
 NEXT\_TASK 120, 267  
 NO\_BLOCK\_LOOP 267  
 NO\_DISTRIBUTE 267  
 NO\_DYNSEL 103, 268  
 NO\_FUSE 74, 268  
 NO\_LOOP\_DEPENDENCE 241, 249, 268  
 NO\_PARALLEL 268  
 NO\_PEEL 83, 268  
 NO\_PROMOTE\_TEST 85, 269  
 NO\_SIDE\_EFFECTS 48, 269  
 NO\_UNROLL 269  
 NO\_UNROLL\_AND\_JAM 269  
 NODE\_PRIVATE 269  
 NODE\_PRIVATE\_POINTER 269  
 OPT\_LEVEL 270  
 ORDERED\_SECTION 270  
 PEEL 83, 271  
 PEEL\_ALL 270  
 PREFER\_FUSE 75, 271  
 PREFER\_PARALLEL 271  
 PROMOTE\_TEST 85, 271  
 PROMOTE\_TEST\_ALL 85, 271  
 ROW\_WISE 272  
 SAVE\_LAST 272  
 SCALAR 272  
 SYNC\_ROUTINE 188, 272  
 TASK\_PRIVATE 272  
 tasking 120  
 THREAD\_PRIVATE 273  
 THREAD\_PRIVATE\_POINTER 273  
 UNROLL 273  
 UNROLL\_AND\_JAM 273  
 compiler optimizations 35  
   options 35  
   overview 8  
 compiler options  
   -align addr 282  
   -align array\_args 280  
   -align cautious 233, 281  
   -align global 282  
   -align no\_addr 282  
   -align no\_global 282  
   -align ptr\_args 282  
   -align restrict\_args 283  
   -align standard 233, 281  
   -align worst 233, 281  
   -align cseries 280  
   -align cti 26, 280  
   -align spp 280  
   -blockloop 72, 278  
   C aliasing 233  
   -cache 276  
   cross compilation 276  
   -ds 103, 277  
   -fl 74  
   -il 283  
   -is 283  
   loop replication 277  
   +max 327  
   +min 327  
   misc. optimization 283  
   misused 229  
   -mrl 59, 277  
   -nds 103, 278  
   -nfl 74  
   -nga 58, 279  
   -ngs 59, 279  
   -no 35, 36, 40, 275  
   -noautopar 127, 276  
   -noblock 72, 278  
   -nopeel 83  
   -noptst 85  
   -nore 128, 283  
   -nosc 39  
   -nsr 86, 284  
   -nuj 80, 277  
   -nur 76, 277  
   -O0 35, 40, 275  
   -O1 35, 45, 275  
   -O2 35, 59, 82, 275  
   -O3 35, 96, 275  
   optimization 35  
   -or 275, 285  
   -or all 276  
   -or array 276  
   -or loop 276  
   -or none 276  
   -or private 276  
   +parallel 327  
   -peel 83, 278  
   -peelall 279  
   -noptst 278  
   -ptst 85, 279  
   -ptstall 85, 279  
   -re 284  
   -sr 86, 284  
   -tm 276  
   -uj 80, 277  
   -ujn 80, 277  
   -uo 53, 54, 257, 284  
   -ur 76, 277  
   -urn 76, 277  
   -Wl, 327  
 compiler parallel support library 301

- accessing 305
- and asymmetric parallelism 303
- and symmetric parallelism 301
- and sync\_routine 325
- asymmetric parallelism example 330
- asymmetric thread functions 308
- barriers 317, 332
- block parallelism example 327
- c\_cond\_lock function 321
- c\_fetch\_and\_add32 function 322
- c\_fetch\_and\_clear32 function 322
- c\_fetch\_and\_dec32 function 322
- c\_fetch\_and\_inc32 function 321
- c\_fetch\_and\_set32 function 322
- c\_fetch32 function 321
- c\_free32 function 320
- c\_init32 function 320
- c\_lock function 320
- c\_unlock function 321
- cps\_barrier function 318
- cps\_barrier\_alloc function 317
- cps\_barrier\_free function 317
- cps\_complex\_cpus function 314
- cps\_complex\_nodes function 314
- cps\_complex\_nthreads function 314
- cps\_is\_parallel function 314
- cps\_ktid 311
- cps\_mutex\_alloc function 318
- cps\_mutex\_free function 318
- cps\_mutex\_lock function 319
- cps\_mutex\_trylock function 319
- cps\_mutex\_unlock function 319
- cps\_node\_cpus function 313
- cps\_node\_id function 313
- cps\_node\_nthreads function 313
- cps\_nthreads function 312
- cps\_plevel function 312
- cps\_ppcall function 306
- cps\_ppcalln function 306
- cps\_std function 311
- cps\_thread\_create function 308
- cps\_thread\_exit function 310
- cps\_thread\_wait function 310
- cps\_topology function 315
- cps\_wait\_attr function 315
- critical sections using low-level functions 336
- cyclic parallelism example 329
  - defined 348
  - finding hypernode ID 313
  - finding kernel thread ID 311
  - finding number of cpus 313
  - finding number of threads 312
  - finding spawn thread ID 311
- high-level synchronization functions 317, 332
- low-level ordered section example 337
- low-level semaphores and critical sections 336
- low-level semaphores and ordered sections 337
- low-level synchronization functions 319, 336
  - m\_cond\_lock function 324
  - m\_fetch\_and\_clear32 function 325
  - m\_fetch\_and\_dec32 function 324
  - m\_fetch\_and\_inc32 function 324
  - m\_fetch32 function 324
  - m\_free32 function 323
  - m\_init32 function 323
  - m\_lock function 323
  - m\_unlock function 323
- mutexes 317, 334
- PARAMS values 306
  - params->max values 306
  - params->min values 306
  - params->node values 306
  - params->threadscope values 307
- setting stack size for spawned threads 308
- spawning symmetric threads 305
- specifying parallelism at compile time 327
- symmetric parallelism examples 327
- thread information functions 311
- thread management functions 305
- compiler pragmas 261
- complex
  - defined 348
- computed statements 87
- concurrent
  - defined 348
- concurrent execution 36
- cond\_lock\_gate function 186
- cond\_lock\_gate\_8 function 186
- conditional induction variable
  - defined 348
- conditionals
  - short-circuit evaluation 38, 39
- constant folding 42
  - defined 348
- constant propagation 42, 45
  - defined 348
- constants
  - strength reduction of 54
- control parallel programming
  - defined 348
- conventional compiler
  - defined 348
- ConvexPVM library 216
  - approaches to parallelism 218
  - CSPP architecture specifier 217
  - intertask communication 217
  - master/slave example 219
  - message buffering 217
  - new hostfile syntax 216
  - SPMD example 223
- copy propagation 49
- counted loop 235
- counter
  - defined 348

CPA  
 defined 349

cps.h 305

CPS\_ANY\_NODE constant 306

cps\_barrier function 318

cps\_barrier\_alloc function 317

cps\_barrier\_free function 317

cps\_complex\_cpus function 314

cps\_complex\_nodes function 314

cps\_complex\_nthreads function 314

CPS\_DIFFERENT\_NODE constant 306

CPS\_GETWAIT constant 316

cps\_is\_parallel function 314

cps\_ktid function 311

cps\_mutex\_alloc function 318

cps\_mutex\_free function 318

cps\_mutex\_lock function 319

cps\_mutex\_trylock function 319

cps\_mutex\_unlock function 319

cps\_node\_cpus function 313

cps\_node\_id function 313

cps\_node\_nthreads function 313

CPS\_NODE\_PARALLEL constant 307

cps\_nthreads function 312

CPS\_PL\_ASYMMETRIC constant 312

CPS\_PL\_NODE constant 312

CPS\_PL\_NONE constant 312

CPS\_PL\_NTHREAD constant 312

CPS\_PL\_PARALLEL constant 312

CPS\_PL\_THREAD constant 312

cps\_plevel function 312

cps\_ppcall function 306

  setting stack size for spawned threads 308

cps\_ppcalln function 306

CPS\_SAME\_NODE constant 306

CPS\_SETWAIT constant 316

CPS\_SETWAITI constant 316

CPS\_SPINWAIT constant 316

CPS\_STACK\_SIZE environment variable

  and CPSlib parallelism 308

  and loop\_parallel 128

  and loop\_private data 129

  and task\_private data 129

cps\_std function 311

CPS\_SUSPEND constant 316

cps\_thread\_create function 308

cps\_thread\_exit function 310

CPS\_THREAD\_PARALLEL constant 307

cps\_thread\_wait function 310

cps\_topology function 315

cps\_wait\_attr function 315

CPSlib

  defined 349

  See compiler parallel support library

CPU

  defined 349

CPU agent

  defined 349

CPU time

  defined 349

CPU-private memory

  defined 349

CPUs

  and thread affinity 101

  minimum/maximum number 1

critical section

  defined 349

critical sections 125, 126, 263

  and gates 192

  and the compiler parallel support library 317

  low-level CPSlib example 336

  manually implemented 206

  multiple 194

  using CPSlib mutexes 334

critical\_section directive and pragma 126, 263

crossbar 11

  defined 349

  illustrated 12

CSPM PVM architecture specifier 217

CSR

  defined 350

CTI 1

CTI interface

  defined 350

CTI ring

  defined 350

CTI rings 2

  illustrated 12

CTIcache 143

  defined 350

  illustrated 13

CTIcache lines 16

  interleaving 26

customer support xxiv

cyclic parallelism

  and CPSlib ordered sections 339

  example 329

---

## D

data alignment

  on CTIcache lines 25

  on natural boundaries 39

data cache 13

  defined 350

data dependency

  defined 350

data localization 59

  and loop replication 59

  benefits 60

  data reuse 66

  defined 350

  inhibitors 86

- preventing 96, 272
- spatial reuse 65
- strip mining 62
- data ordering 39
- data parallel programming
  - defined 350
- data privatization
  - in parallel loops 266
  - in tasks 272
- data privitytization
  - and prefer\_parallel 117
  - in parallel tasks 136
  - loop 129
- data reuse 65
  - defined 367
  - example 66
  - spatial 65
  - temporal 66
- data type
  - defined 350
- dead code
  - eliminating 49
- deadlock
  - defined 351
- dependencies
  - apparent 241
  - C example 241
  - Fortran example 240
  - hidden 246
  - ignoring 268
  - isolating with ordered sections 198
  - loop carried 87, 240
  - ordering 270
  - reductions 106, 248
  - unordered 125, 263
- Dependencies column
  - in array table 291
- dependency 240
- determined order of execution 252
- direct mapped cache
  - defined 346
- direct memory access
  - defined 351
- directives
  - ALIGN\_CTI 25, 262
  - BARRIER 184, 262
  - BEGIN\_TASKS 262
  - BLOCK\_LOOP 71, 263
  - BLOCK\_SHARED 263
  - CRITICAL\_SECTION 125, 192, 263
  - DYNSEL 103
  - END\_CRITICAL\_SECTION 125, 192, 264
  - END\_ORDERED\_SECTION 193, 264
  - END\_TASKS 264
  - FAR\_SHARED 264
  - FAR\_SHARED\_POINTER 265
  - form 261
  - Fortran compiler 261
  - GATE 184, 265
  - loop blocking 71
  - LOOP\_PARALLEL 112, 265
  - LOOP\_PARALLEL(ORDERED) example 191
  - LOOP\_PRIVATE 130, 266
  - memory class 148
  - misused 240
  - NEAR\_SHARED 266
  - NEAR\_SHARED\_POINTER 267
  - NEXT\_TASK 267
  - NO\_BLOCK\_LOOP 71, 267
  - NO\_DISTRIBUTE 64, 267
  - NO\_DYNSEL 103, 268
  - NO\_FUSE 74
  - NO\_LOOP\_DEPENDENCE 90, 106, 268
  - NO\_PARALLEL 107, 268
  - NO\_PEEEL 268
  - NO\_PROMOTE\_TEST 85, 269
  - NO\_SIDE\_EFFECTS 269
  - NO\_UNROLL 269
  - NO\_UNROLL\_AND\_JAM 80, 269
  - NODE\_PRIVATE 269
  - NODE\_PRIVATE\_POINTER 269
  - ORDERED\_SECTION 193, 270
  - PEEL 83, 271
  - PEEL\_ALL 83, 270
  - PREFER\_FUSE 75, 271
  - PREFER\_PARALLEL 112, 271
  - PROMOTE\_TEST 85, 271
  - PROMOTE\_TEST\_ALL 85, 271
  - ROW\_WISE 272
  - SAVE\_LAST 138, 272
  - SCALAR 96, 272
  - sync\_routine 188, 272
  - TASK\_PRIVATE 136, 272
  - THREAD\_PRIVATE 273
  - THREAD\_PRIVATE\_POINTER 273
  - UNROLL 77, 273
  - UNROLL\_AND\_JAM 80
- Dist entry
  - in optimization report 287
- distributed memory
  - defined 351
- distributed part
  - defined 351
- distribution
  - loop 63
- DoD 342
- double
  - defined 351
- doubleword
  - defined 351
- ds option 103, 277
- dummy argument
  - defined 351
- dynamic memory

- and memory class pointers 155
- and memory\_class\_malloc 156
- assigning block\_shared class 169
- assigning classes 158
- assigning far\_shared class 168
- assigning near\_shared class 162
- assigning node\_private class 159
- assigning thread\_private class 158
- class assignments 155
- default classes 157
- dynamic memory class assignments 155
  - and wrong answers 242
  - incorrect pointer use 245
- dynamic selection 102
  - and dynsel directive and pragma 263
  - and reentrancy 128
  - defined 352
  - in optimization report 293
  - workload-based 102
- DYNSEL compiler directive 103
- dynsel compiler directive and pragma 263
- dynsel directive and pragma 103
- DynSel entry
  - in optimization report 287

---

## E

- elimination
  - of common subexpressions 42, 50
  - of dead code 49
  - of redundant uses 42
  - of type conversions 258
- encache
  - defined 352
- end\_critical\_section directive and pragma 126, 264
- end\_ordered\_section directive and pragma 264
- END\_TASKS directive 120
- end\_tasks directive and pragma 264
- entries
  - multiple routine 87, 104
- environment variables
  - CPS\_STACK\_SIZE 128, 308
  - PVMFRAGSIZE 217
- error message
  - overflow 43
- evaluation order 250
- exception
  - defined 352
- executables
  - modifying attributes via mpa 314
- execution stream
  - defined 352
- Exemplar system overview 1
- exits
  - multiple routine 87, 104
- expressions

- equivalent 257
- ext option
  - and aliasing 231

---

## F

- far\_shared directive 264
- far\_shared memory 147
  - dynamic allocation 168
  - static assignments 154
- far\_shared memory class 264
- far\_shared\_pointer directive 265
- fault
  - defined 352
- Federal Information Processing (FIPS) 342
- .fil file 283
  - creating 283
- file utility 327
- fl option 74
- floating point
  - defined 352
- floating point imprecision
  - example 238
- floating-point
  - imprecision 54, 229
- floating-point imprecision 239
- FLOPS
  - defined 352
- FMPYADD instruction 80
- folding
  - constant 45
  - of constants 42
- Footnoted Iter. Var. column
  - in variable name footnote table 290
- footnotes
  - in optimization report 290
  - in optimization report, example 293
- for loops
  - specifying induction variables for parallelization 114
- formal parameter
  - defined 352
- Fortran
  - defined 353
- Fortran 90
  - defined 353
- Fortran 90 array expressions
  - and parallelization 101
- forward LCDs 104
- free\_barrier function 186
- free\_barrier\_8 function 186
- free\_gate function 186
- free\_gate\_8 function 186
- fully associative cache
  - defined 346
- function
  - defined 353

- functional block 12
  - illustrated 13
- functional unit
  - defined 353
- functional units 36
- functions
  - parallelization of intrinsic 107
- fusion
  - of loops 72

---

## G

- gate
  - defined 353
- gate\_t data type 184
- gate8\_t data type 184
- gates 184, 265
  - allocating 185
  - deallocation 186
  - in C 184
  - in Fortran 184
  - locking 186
  - unlocking 187
- Gbyte
  - defined 353
- gigabyte
  - defined 353
- global
  - variable aliasing 236
- global memory
  - defined 354
- global optimization
  - defined 353
- global register allocation 56
  - and compile time 59
  - and scalar replacement 86
- global variable
  - defined 354
- global variable aliasing 231
- global virtual memory
  - defined 354
- globally shared memory
  - defined 353
- globally shared memory (GSM) 3
- GRA 56
- granularity
  - defined 354
- GSM (globally shared memory) 3

---

## H

- hand-rolled loop
  - defined 354
- hand-rolled loops

- manually parallelizing 132
- hidden alias
  - defined 354
- hidden aliases 230
- hidden dependencies 246
  - Fortran example 247
- hidden ordered sections 253
  - Fortran example 254
- High Performance Fortran (HPF)
  - defined 354
- high-level synchronization functions 317
- hoist
  - defined 354
- host
  - defined 355
- hostfile
  - for ConvexPVM 216
  - specifying subcomplexes 216
- HP
  - defined 355
- HP-UX
  - defined 355
- hypercube
  - defined 355
- hypernode 1
  - defined 355
  - finding logical ID using compiler parallel support library 313
  - finding logical ID using my\_node() function 180
  - illustrated 12
- hypernode-local memory 14
- hypernodes
  - finding available using compiler parallel support library 314
  - finding number of active threads using compiler parallel support library 313
  - finding thread topology 315
  - minimum/maximum number 1

---

## I

- Id Num.
  - in analysis table 289
  - in loop table 285
- Id Num. column
  - in privatization table 290
- idle threads
  - setting wait attributes 315
  - specifying states 315
  - spin-waiting 100
  - states of 100
  - suspended 100
- IF tests
  - short circuiting 39
- IF-DO optimizations 81
- if-for optimizations 81

---

- options 278
- inhibitors of localization
  - I/O statements 95
- il option 283
- incorrect answers
  - and array pointers 245
  - and evaluation order 250
  - and floating point imprecision 237
  - and hidden dependencies 246, 247
  - and incrementing by zero 251
  - and large trip counts 253
  - and misused directives 240
  - and misused memory classes 242
  - and no\_loop\_dependence 249
  - and ordered sections 253
  - and parallel execution 252
  - and parallelism 254
- incrementing by zero 251
  - examples 252
- induction variable 235
- induction variables 54
  - and parallelization directives 113
  - conversion 258
  - in parallel hand-rolled loops 132
  - indicating to compiler 132
    - primary 132
    - privatizing secondary 133
    - secondary 133
- inhibitors of localization 86
  - aliasing 91
  - GOTO statements 94
  - loop-carried dependencies 87
  - procedure calls 95
  - RETURN/STOP statements 94
- inhibitors of localizatoin
  - multiple entries/exits 93
- inhibitors of parallelization 104
  - loop-carried dependencies 104
- inline substitution 283
- inlining
  - defined 355
  - using -il 283
- Institute for Electrical and Electronic Engineers(IEEE)
  - defined 355
- instruction
  - defined 356
- instruction cache 13
  - defined 356
- instruction mnemonic
  - defined 356
- instructions
  - scheduling 36, 40
  - span-dependent 36
- integral division
  - defined 356
- interchange
  - loop 64

- Interchange entry
  - in optimization report 287
- interconnect cache 14
- interface
  - defined 356
- interleaved memory
  - defined 356
- interleaving 26
  - example 28
- interprocedural optimization
  - defined 356
- interprocessor communication
  - defined 356
- intrinsic
  - defined 356
- intrinsic functions
  - parallelization of 107
- invalid subscripts 237
- invariant expressions 257
- is option 283
- Iter. Var.
  - in analysis table 289
  - in loop table 286
- Iter. Var. column
  - in privatization table 290
- iteration variables 251

---

## J

- job scheduler
  - defined 356
- join
  - defined 356
- joins
  - and cps\_ppcall 307
  - and CPSlib asymmetric threads 303
  - and CPSlib symmetric parallelism 303
  - as implicit barrier 332
- jump
  - defined 357

---

## K

- kbyte
  - defined 357
- kernal thread ID 182
- kernal thread identifier
  - defined 372
- kernel
  - defined 357
- kernel thread ID
  - finding using compiler parallel support library 311
- kilobyte
  - defined 357

ktid  
  defined 357

---

## L

large application  
  defined 357

large trip counts 253

latency  
  defined 357

level of parallelism  
  finding 180

level\_of\_parallelism() function 180

libpvm  
  defined 357

limits of optimization 229

Line Num.

  in loop table 285

Line Num. column  
  in analysis table 289

  in array table 291

  in privatization table 290

  in test table 289

linker  
  defined 357

load  
  defined 357

load balancing  
  and logical hypernode ID 180

local optimization  
  defined 358

local variable aliasing 231

locality of reference  
  defined 357

localization  
  of data 59  
  preventing 272

lock\_gate function 186

lock\_gate\_8 function 186

logical address  
  defined 358

logical hypernode ID  
  finding using CPSlib 313

logical hypernode ID  
  finding using my\_node() function 180

loop  
  counter pointer 235

loop blocking 65  
  block\_loop directive 263

  compiler options 278

  data reuse 65

  defined 358

  example 67

  illustration 68

  matrix multiply example 70

  no\_block\_loop directive and pragma 267

  related directives and pragmas 71

  spatial reuse 65

  temporal reuse 66

loop carried dependence  
  defined 358

loop carried dependencies 86, 87  
  and parallelization 104

  apparent 106

  backward 105

  forward 104

  no\_loop\_dependence directive and pragma 268

  ordering 270

  output 105

  unordered 263

loop carried dependency  
  illustrated 89

loop cloning 102

loop constant  
  defined 358

loop distribution 63  
  defined 359

  in optimization report 293

  no\_distribute directive and pragma 267

  parts in optimization report 293

loop fusing 78

loop fusion 72

  and Fortran 90 array assignments 74

  and no\_fuse directive 268

  and other transformations 74

loop ID number

  in optimization report 287

loop independent dependence (LID)  
  defined 359

loop induction variable  
  defined 359

loop interchange 64  
  defined 359

  in optimization report 293

loop invariant computation  
  defined 359

loop jamming 78

loop parallelization  
  defined 364

loop peeling 82

  and loop replication 83

  compiler options 278

  disabling 278

  enabling 278, 279

  in optimization report 297

  peel directive and pragma 271

  peel\_all directive and pragma 270

  preventing 268

loop private data

  save\_last directive and pragma 138, 272

loop replication 279

  defined 359

loop replication compiler options 277

- loop unroll and jam 77
  - compiler options 80
  - matrix multiply example 78
  - options 277
- loop unrolling 75
  - and jamming 77
  - compiler options 277
  - depth 75
  - factor 75
  - partial 75
  - total 75
  - unroll directive and pragma 273
  - using -ur 277
- loop\_parallel directive and pragma 112, 118, 265
- loop\_parallel(chunk\_size) directive and pragma 112
- loop\_parallel(ivar) directive and pragma 113
- loop\_parallel(max\_threads) directive and pragma 112
- loop\_parallel(nodes) directive and pragma 112
- loop\_parallel(ordered) directive and pragma 113, 190
- loop\_parallel(threads) directive and pragma 112
- loop\_private directive
  - Fortran example 130
- loop\_private directive and pragma 129, 266
  - and CPS\_STACK\_SIZE 129
- loop-carried dependency (LCD) 87, 240
  - forward 104
- loops with calls
  - parallelizing 118
- low-level synchronization functions 319

---

## M

- m\_cond\_lock function 324
- m\_fetch\_and\_clear32 function 325
- m\_fetch\_and\_dec32 function 324
- m\_fetch\_and\_inc32 function 324
- m\_fetch32 function 324
- m\_free32 function 323
- m\_init32 function 323
- m\_lock function 323
- m\_unlock function 323
- machine exception
  - defined 359
- main memory
  - defined 360
- main procedure
  - defined 360
- main program
  - defined 360
- manual synchronization 204
- massively parallel processing (MPP) 1
- massively parallel processor
  - defined 360
- master/slave ConvexPVM example 219

- matrix multiplication 64
- matrix multiply
  - blocking example 70
- +max option 327
- max\_threads
  - attribute to loop\_parallel 112
- Mbyte
  - defined 360
- megabyte
  - defined 360
- megaflops
  - defined 360
- memory
  - physical 14
  - private vs. shared 144
  - subcomplex 33
  - virtual 14
- memory class pointers 155
  - C 156
  - Fortran 155
  - private pointers with shared data 156
- memory classes 143
  - acceptable pointer/data class combinations 156
  - and spp\_prog\_model.h 149, 177
  - and suitable pointer classes 157
  - assigning in C 149
  - assigning in Fortran 148
  - assignments 148
  - block\_shared 148, 263
  - default for dynamic allocation 157
  - dynamic assignments 155
  - dynamically assigning block\_shared 169
  - dynamically assigning far\_shared 168
  - dynamically assigning near\_shared 162
  - dynamically assigning node\_private 159
  - dynamically assigning thread\_private 158
  - far\_shared 147
  - far\_shared directive 264
  - far\_shared\_pointer directive 265
  - incorrect use 242
  - incorrect use examples 242
  - near\_shared 147
  - near\_shared\_pointer 266, 267
  - node\_private 146, 269
  - node\_private\_pointer 269
  - physical addressing illustrated 145
  - pointers 155
  - static assignments 150
  - static far\_shared assignments 154
  - static near\_shared assignments 153
  - static node\_private assignments 151
  - static thread\_private assignments 150
  - thread\_private 146, 273
  - thread\_private\_pointer 273
  - virtual addressing illustrated 144
  - virtual to physical mapping 144
- memory configurations 2

- memory management
  - defined 360
- memory type of stack
  - finding 181
- memory\_class\_malloc function 156
- memory\_type\_of\_stack() function 181
- message
  - defined 360
  - overflow error 43
- message passing 7, 215
  - defined 360
  - features 215
  - library 216
  - master/slave example 219
  - parallelism of programs 215
  - SFMD example 223
- message passing/shared memory hybrids 7
- MIMD
  - defined 361
- +min option 327
- miscellaneous optimization options 283
- misused directives 229
- moving code 52, 257
- mpa utility 314, 327
- MPP (massively parallel processing) 1
- mrl option 59, 277
  - and dynamic selection 104
  - and global register allocation 59
  - and loop blocking 72
  - and loop peeling 83
  - and test promotion 85
  - and unroll and jam 81
  - and unrolling 77
- multiple routine entries 87, 104
- multiprocessing
  - defined 361
- mutex
  - defined 361
- mutexes
  - allocating with cps\_mutex\_alloc function 318
  - conditionally acquiring with cps\_mutex\_trylock function 319
  - CPSlib example 334
  - defined 317
  - freeing with cps\_mutex\_free function 318
  - locking with cps\_mutex\_lock function 319
  - unlocking with cps\_mutex\_unlock function 319
- mutual exclusion
  - defined 361
- mutual exclusion areas
  - CPSlib example 334
- my\_node() function 180
- my\_thread() function 179

---

## N

- natural data type boundaries
  - alignment on 39, 280
- nds option 103, 278
- near\_shared directive 266
- near\_shared memory 147
  - dynamic allocation 162
- near\_shared\_pointer directive 267
- near-shared memory
  - defined 361
  - static assignments 153
- negate
  - defined 361
- network
  - defined 361
- new loops
  - in loop table 287
- NEXT\_TASK directive 120
- next\_task directive and pragma 267
- nfl option 74
- nga compiler option 58, 279
- ngs compiler option 59, 279
- no option 35, 36, 40, 275
- no\_block\_loop directive and pragma 71, 267
- no\_distribute directive and pragma 64, 267
- no\_dynsel directive and pragma 103, 268
- no\_fuse compiler directive and pragma 74
- no\_fuse directive and pragma 268
- NO\_LOOP\_DEPENDENCE directive 241
- no\_loop\_dependence directive and pragma 90, 249, 268
  - and apparent dependencies 241
  - improper use 249
- no\_parallel directive and pragma 107, 268
- NO\_PEEEL directive 83
- no\_peel directive and pragma 268
- NO\_PROMOTE\_TEST directive 85
- no\_promote\_test directive and pragma 269
- NO\_SIDE\_EFFECTS directive 48
- no\_side\_effects directive and pragma 269
  - caution 49
- no\_unroll directive and pragma 269
- no\_unroll\_and\_jam directive and pragma 80, 269
- noautopar option 127, 276
- noblock compiler option 72
- noblock option 278
- node
  - defined 361
- node parallelism
  - specifying for loops 112
- node parallelization
  - specifying for tasks 120
- node\_private directive 269
- node\_private memory 146
  - dynamic allocation 159

- incorrect pointer use example 246
- incorrect use example 244
- static assignments 151
- node\_private\_pointer directive 269
- node-private memory 361
- nodes
  - attribute to loop\_parallel 112
- nonblocking crossbar
  - defined 362
- nondeterminism
  - parallel 252
- non-uniform memory access (NUMA)
  - defined 362
- nopeel option 83, 278
- noptst option 85, 278
- nore option 128, 283
- nosc option 39
- notational conventions xxii
- nsr option 86, 284
- nuj option 80, 277
- num\_node\_threads() function 179
- num\_nodes() function 179
- num\_procs() function 178
- num\_threads() function 178
- number of hypernodes
  - finding 178
- number of processors
  - finding 178
- number of threads
  - finding 178
- number of threads on hypernode
  - finding 179
- nur option 76, 277

---

## O

- O0 option 35, 40, 275
- O1 option 35, 45, 275
  - and invariant code 230
- O2 option 35, 59, 82, 275
- O3 option 35, 96, 275
- offset
  - defined 362
- opcode
  - defined 362
- operating system
  - defined 362
- opt\_level pragma 270
- optimization 236
  - basic-block 45
  - defined 362
  - global 45
  - IF-DO 81
  - limits of 229
  - machine-dependent 45
  - potentially unsafe 284
  - report 234, 285
  - unsafe 53, 54
- Optimization column
  - in array table 291
- optimization level
  - and opt\_level pragma 270
  - defined 362
- optimization options
  - cache 276
  - noblock 278
  - or 275
- optimization report 276, 285
  - \* entry 287
  - analysis column 289
  - analysis table 289
  - Blocked entry 288
  - contents 285
  - Dist entry 287
  - DynSel entry 287
  - ID number column 285, 289
  - Inter entry 287
  - Interchange entry 288
  - iteration variable column 286, 289, 290
  - line number column 285
  - loop peeling 297
  - nested loop example 291
  - new loops 298
  - new loops column 287
  - optimizing/special transformations column 287
  - PARALLEL entry 287
  - Pattern entry 288
  - Peel entry 287
  - Promote entry 287
  - Reduction entry 288
  - Removed entry 288
  - reordering transformation column 286
  - Scalar entry 287
  - single loop example 295
  - StripMine entry 288
  - test table 289
  - Unroll entry 288
  - variable name footnotes 290
- optimizing/special transformations
  - in loop table 287
- or option 276, 285
- order of evaluation 250
- ordered
  - attribute to loop\_parallel 113
- ordered parallelism 190, 191
  - example 191
- ordered parallelization
  - defined 364
- ordered sections 197, 270
  - and dependencies 198
  - and gates 193
  - hidden 253
  - low-level CPSlib example 337

- ordered task parallelization 120
- ordered\_section directive and pragma 270
- ordering
  - of data for proper alignment 40
- output LCDs 105
- overflow 257
  - error message 43
- overhead
  - parallelization 102
- oversubscript
  - defined 363
- oversubscripting 237
- oversubscription
  - defined 363

---

## P

- packet
  - defined 363
- page
  - defined 363
- page fault
  - defined 363
- page frame
  - defined 363
- PARALLEL entry
  - in optimization report 287
- parallel optimization
  - defined 363
- +parallel option 327
- parallel strip length
  - defined 370
- parallel tasks
  - begin\_tasks 262
  - end\_tasks directive and pragma 264
  - next\_task directive and pragma 267
- Parallel Virtual Machine (PVM)
  - defined 363
- parallel virtual machine (PVM) library 216
- parallelism
  - and CPSlib barriers 332
  - and CPSlib programs 327
  - and loop induction variables 113
  - asymmetric example 330
  - block 327
  - chunk-based 114
  - cyclic 329
  - default 97, 100
  - disabling automatic 276
  - finding level of using compiler parallel support library 312
  - ordered example 191
  - simple example 97
  - stride-based 114
  - strip-based 99
  - using compiler parallel support library to determine

- presence 314
- parallelization
  - and Fortran 90 constructs 101
  - asymmetric, using CPSlib 303
  - automatic implementation 99
  - basic operation 96
  - begin\_tasks directive and pragma 262
  - by chunks 112
  - defined 363
  - in optimization report 293
  - inhibitors of 104
  - limitations 119
  - loop\_parallel directive and pragma 265
  - loop\_private directive and pragma 266
  - maximum threads in a loop 112
  - next\_task directive and pragma 267
  - nondeterministic execution 252
  - of for loops 114
  - of Fortran loops 114
  - of loops with calls 118
  - optimization level -O3 96
  - optimization overview 9
  - optimizations 102
  - ordered 113, 190
  - ordering dependencies 270
  - prefer\_parallel directive and pragma 271
  - preventing 107, 268, 272
  - simple manual loop 112
  - simple manual task 120
  - specifying node 112
  - specifying threads 112
  - stride-based 115
  - symmetric, using CPSlib 301
  - task node-way 120
  - task thread-way 120
  - task\_private directive and pragma 272
- parallelization directives
  - list 108
- parallelization overhead 102
- parameter
  - actual 341
  - defined 364
- parentheses
  - use of 250
- PA-RISC
  - defined 363
- PA-RISC 7100 2
- partial loop unrolling 76
- path
  - defined 364
- pcc option
  - and aliasing 231
- PEEL directive 83
- peel directive and pragma 83, 271
- Peel entry
  - in optimization report 287
- peel option 83, 278

- peel\_all compiler directive and pragma 270
- PEEL\_ALL directive 83
- peelall option 279
- peeling
  - loop 82
- physical address
  - defined 365
- physical address space
  - defined 365
- physical hypernode ID 180, 313
- physical memory 14
  - access times 13
  - and subcomplexes 33
  - classes 143
  - configuration 3
  - configurations 2
  - defined 365
  - GSM 3
  - hypernode local 14
  - interconnect cache 14
  - interleaving 26
  - maximum 2
  - minimum 2
  - partitioning 14
  - subcomplex-global 14
- pipeline
  - defined 365
- pipelining 36
  - defined 365
- pointer
  - loop counter 235
- pointer tracking
  - defined 365
- pointers
  - default memory classes 157
  - memory class 155
  - memory class in Fortran 155
  - memory\_class\_malloc form 156
  - to shared data 156
- potentially unsafe optimizations 284
- pragmas
  - align\_cti 26, 262
  - begin\_tasks 262
  - block\_loop 71, 263
  - C compiler 261
  - critical\_section 126, 192, 263
  - dynsel 103, 263
  - end\_critical\_section 126, 192, 264
  - end\_ordered\_section 193, 264
  - end\_tasks 264
  - form 261
  - gate 265
  - loop blocking 71
  - loop\_parallel 112, 265
  - loop\_private 130, 266
  - misused 240
  - next\_task 267
  - no\_block\_loop 71, 267
  - no\_distribute 64, 267
  - no\_dynsel 103, 268
  - no\_fuse 75, 268
  - no\_loop\_dependence 90, 242, 268
  - no\_parallel 107, 268
  - no\_peel 268
  - no\_promote\_test 269
  - no\_side\_effects 269
  - no\_unroll 269
  - no\_unroll\_and\_jam 80, 269
  - opt\_level 270
  - ordered\_section 193, 270
  - peel 271
  - peel\_all 270
  - prefer\_fuse 75, 271
  - prefer\_parallel 112, 271
  - promote\_test 271
  - promote\_test\_all 271
  - save\_last 138, 272
  - scalar 96, 272
  - sync\_routine 188, 272
  - task\_private 136, 272
  - unroll 77, 273
  - unroll\_and\_jam 80
- prefer\_fuse compiler directive and pragma 271
- prefer\_fuse compiler directive and pragma 75
- prefer\_parallel directive and pragma 112, 117, 271
- Priv. Var. column
  - in privatization table 290
- private data objects 15
- privatization
  - of secondary induction variables 133
- privatization information
  - in optimization report 290
- privatization table 290
- procedure
  - defined 365
- procedure calls
  - parallelizing 118
- process
  - defined 365
- process memory
  - defined 365
- processor cache line 16
- processor functional units 36
- program unit
  - defined 366
- programming model 6
  - defined 366
  - message passing 7
  - message passing/shared memory hybrids 7
  - shared memory 6
- Promote entry
  - in optimization report 287
- PROMOTE\_TEST directive 85
- promote\_test directive and pragma 271

PROMOTE\_TEST\_ALL directive 85  
promote\_test\_all directive and pragma 271  
promotion  
  test 84  
propagating constants 42, 45  
propagating copies 49  
-ptst option 85, 279  
-ptstall option 85, 279  
PVM  
  defined 363, 366  
PVMFRAGSIZE environment variable 217

---

## Q

queue  
  defined 366

---

## R

rank  
  defined 366  
-re option 284  
read  
  defined 366  
REAL variables  
  effect of 54  
recurrence  
  defined 366  
recursion  
  defined 366  
recursive call  
  defined 366  
reduced instruction set computer (RISC)  
  defined 366  
reduction  
  as dependency 248  
  C example 249  
  defined 366  
  Fortran example 248  
  parallelizable 106  
  strength 53, 54  
  tree height 37, 38  
redundant loads  
  elimination of 41  
redundant-assignment elimination 40, 46  
redundant-test elimination 81, 82  
redundant-use elimination 42  
reentrancy  
  defined 367  
reentrant compilation 127, 283  
  and loop replication 128  
reentrant procedure  
  defined 48  
reference 367

register  
  defined 367  
register allocation 36  
registers  
  global allocation of 56  
reordering transformation  
  and incorrect answers 250  
  in loop table 286  
replication  
  of loops 277, 279  
report  
  optimization 276  
RISC  
  defined 366  
-rl option 279  
rounding  
  defined 367  
roundoff error 229, 239, 257  
ROW\_WISE directive 272  
row-major array storage 272  
row-major order  
  defined 368

---

## S

save\_last directive and pragma 138, 272  
Scalable Coherent Interface 2  
scalable coherent interface  
  defined 368  
scalable parallel processing (SPP) 1  
scalable parallel processor  
  defined 368  
scalar directive and pragma 96, 272  
Scalar entry  
  in optimization report 287  
scalar replacement 85  
  and global register allocation 86  
scheduling  
  instructions 36, 40  
SCI 2  
  defined 368  
scope  
  defined 368  
Secondary 133  
semaphore  
  defined 368  
semaphores  
  acquiring cache-based 320  
  allocating cache-based for synchronization 320  
  allocating memory-based 323  
  and compiler parallel support library functions 319  
  conditionally acquiring cache-based 321  
  conditionally locking memory-based 324  
  CPSlib low-level and critical sections 336  
  CPSlib low-level and ordered sections 337  
  freeing cache-based 320

- freeing memory-based 323
- locking memory-based 323
- unlocking cache-based 321
- unlocking memory-based 323
- serial execution
  - specifying for a loop 272
- shape
  - defined 368
- shared data objects 15
- shared memory
  - advanced programming example 209
  - basic programming 111
- shared memory address space 14
- shared memory programming 6
  - advanced 177
- shell
  - defined 369
- short-circuit evaluation of conditionals 38, 39
- side effects
  - ignoring 269
- SIMD
  - defined 369
- simplification
  - algebraic 44
  - trigonometric 44
- single
  - defined 369
- SMP
  - defined 371
- socket
  - defined 369
- socket pair
  - defined 369
- source code
  - defined 369
- source file
  - defined 369
- space
  - defined 369
- span-dependent instructions 36
- spatial reference
  - defined 369
- spatial reuse 65
  - defined 367
  - illustration 69
- spawn
  - defined 369
- spawn context 312
  - defined 369
- spawn thread ID 183
  - finding using compiler parallel support library 311
- spawn thread identifier
  - defined 372
- spawn\_sym\_t structure
  - declared 306
- spawning
  - and compiler parallelism 99
    - using CPSlib 306
- spin-waiting 100
- SPMD
  - defined 370
- SPMD ConvexPVM example 223
- SPP
  - defined 368
- SPP (scalable parallel processing) 1
- spp\_prog\_model.h 149, 177
- sr option 86, 284
- stack
  - defined 370
  - finding memory class of 181
  - for spawned threads 128
  - setting size for spawn threads 128, 308
- statements
  - ASSIGN 47
- static memory class assignments 150
- std option
  - and aliasing 231
- stid
  - defined 370
- stop value 235, 236
  - global variable 236
  - variable aliasing 235
- store
  - defined 370
- strength reduction 257
  - arithmetic 53
  - at -O1 53
  - of induction variables 54
- stride-based parallelism 114
- stride-based parallelization
  - defined 364
- stride-parallelized loop
  - example 115
- strip mining 62
  - defined 370
  - in optimization report 293
- strip-based parallelism 99, 100, 114
- strip-based parallelization
  - defined 364
- strip-mining
  - conceptual example 99
- subcomplex
  - defined 371
- subcomplexes 31
  - finding number of active threads using compiler parallel support library 314
  - illustrated 32
  - memory 33
  - physical configuration 31
- subcomplex-global memory 14
- subexpressions
  - elimination of 42, 50
- subroutine
  - defined 371

- subscripts
  - invalid 237
- substitution
  - of assignments 41
- superscalar
  - defined 371
- support
  - technical xxiv
- suspended threads 100
- symmetric multiprocessor
  - defined 371
- symmetric parallelism 301
  - block 327
  - cyclic 329
  - examples 327
- symmetric threads
  - compiler parallel support library spawn functions 305
- sync\_routine directive and pragma 188, 272
  - and CPSlib 325
- synchronization
  - and critical sections 193
  - barriers 184
  - CPSlib high-level functions 332
  - defined 371
  - denoting CPSlib routines 325
  - denoting routines 188
  - functions 185
  - gates 184
  - high-level functions 317
  - in ordered parallel loops 198
  - low level functions 319
  - low-level CPSlib functions 336
  - manual 204
  - tools for 184
- synchronizing code 184
- sysadmin
  - defined 371
- system administrator
  - defined 371
- system manager
  - defined 371
- system organization 11
- system subcomplex
  - defined 371
- specifying maximum threads 120
- task private data 136, 272
- task\_private directive
  - C example 137
- task\_private directive and pragma 136, 272
  - and CPS\_STACK\_SIZE 129
- tasking directives 120, 121
- temporal reuse 66
  - defined 367
- term
  - defined 371
- test elimination
  - redundant 81
- test promotion 84
  - and loop replication 85
  - disabling 278
  - enabling 279
  - in optimization report 289, 298
  - preventing 269
  - promote\_test directive and pragma 271
  - unlimited 271
- test table
  - analysis column 289
  - column number column 289
  - in optimization report 289
  - line number column 289
  - test transformation column 289
- test transformation column
  - in test table 289
- thread
  - defined 96, 371
  - idle states 100
- thread affinity 101
- thread ID
  - finding 179
- thread identifier
  - defined 372
- thread IDs 182
  - and my\_thread() 183
  - and num\_nodes() 183
  - and num\_threads() 182
  - assignments 182
  - kernal 182
  - num\_node\_threads() 183
  - spawn 183
- thread management functions
  - in compiler parallel support library 305
- thread parallelism
  - specifying for loops 112
- thread parallelization
  - specifying for tasks 120
- thread\_private directive 273
- thread\_private memory 146
  - dynamic allocation 158
  - incorrect use 243
  - static assignments 150
- thread\_private\_pointer directive 273

## T

- target machine
  - specifying 276
- target machine option 276
- task list
  - defined 120
- task parallelization
  - defined 364
  - examples 122
  - ordered 120

thread-private  
  defined 372  
threads  
  attribute to loop\_parallel 112  
  finding topology on hypernodes 315  
  setting wait attributes 315  
  suspended 100  
-tm option 276  
topology  
  of threads on hypernodes 315  
traditional parallel computers 2  
tree  
  balancing 37  
tree-height reduction 37, 38  
  defined 372  
trip count  
  defined 372  
  overflow 253  
type conversions  
  eliminating 257, 258  
  of constants 43  
typographic conventions xxii

---

## U

-uj option 80, 277  
-ujn option 80  
unbalanced tree 37  
unlock\_gate function 187  
unlock\_gate\_8 function 187  
unroll and jam  
  and loop replication 80  
  compiler options 80, 277  
  of loops 77  
unroll compiler directive and pragma 77  
UNROLL directive 277  
unroll directive and pragma 273  
unroll\_and\_jam compiler directive and pragma 80  
unrolling  
  and loop replication 77  
  depth 75  
  factor 75  
  loop 75, 277  
unsafe optimizations 53, 54  
  potential 54  
unsigned  
  defined 373  
-uo option 53, 54, 257, 284  
  example 258  
-ur option 76, 277  
-urn option 76, 277  
user interface  
  defined 373  
User Variable Name column  
  in variable name footnote table 290  
utility

defined 373

---

## V

value  
  iteration 251  
Var. Name column  
  in array table 291  
variable privatization  
  in optimization report 290  
  loop example 130  
  loop\_private directive and pragma 129  
  of secondary induction variables 133  
  saving last values 138  
  task example 137  
  task\_private directive and pragma 136  
variables  
  abbreviated in optimization report 290  
  footnoted in optimization report 293  
  in EQUIVALENCE 86, 104  
  induction 54, 258  
vector  
  defined 373  
vector processor  
  defined 373  
virtual address  
  defined 373  
virtual address space 14  
virtual aliases  
  defined 373  
virtual machine  
  defined 373  
virtual memory 14  
  and subcomplexes 33  
  block\_shared 15  
  classes 14  
  defined 373  
  far\_shared 15  
  near\_shared 15  
  node\_private 15  
  thread\_private 15

---

## W

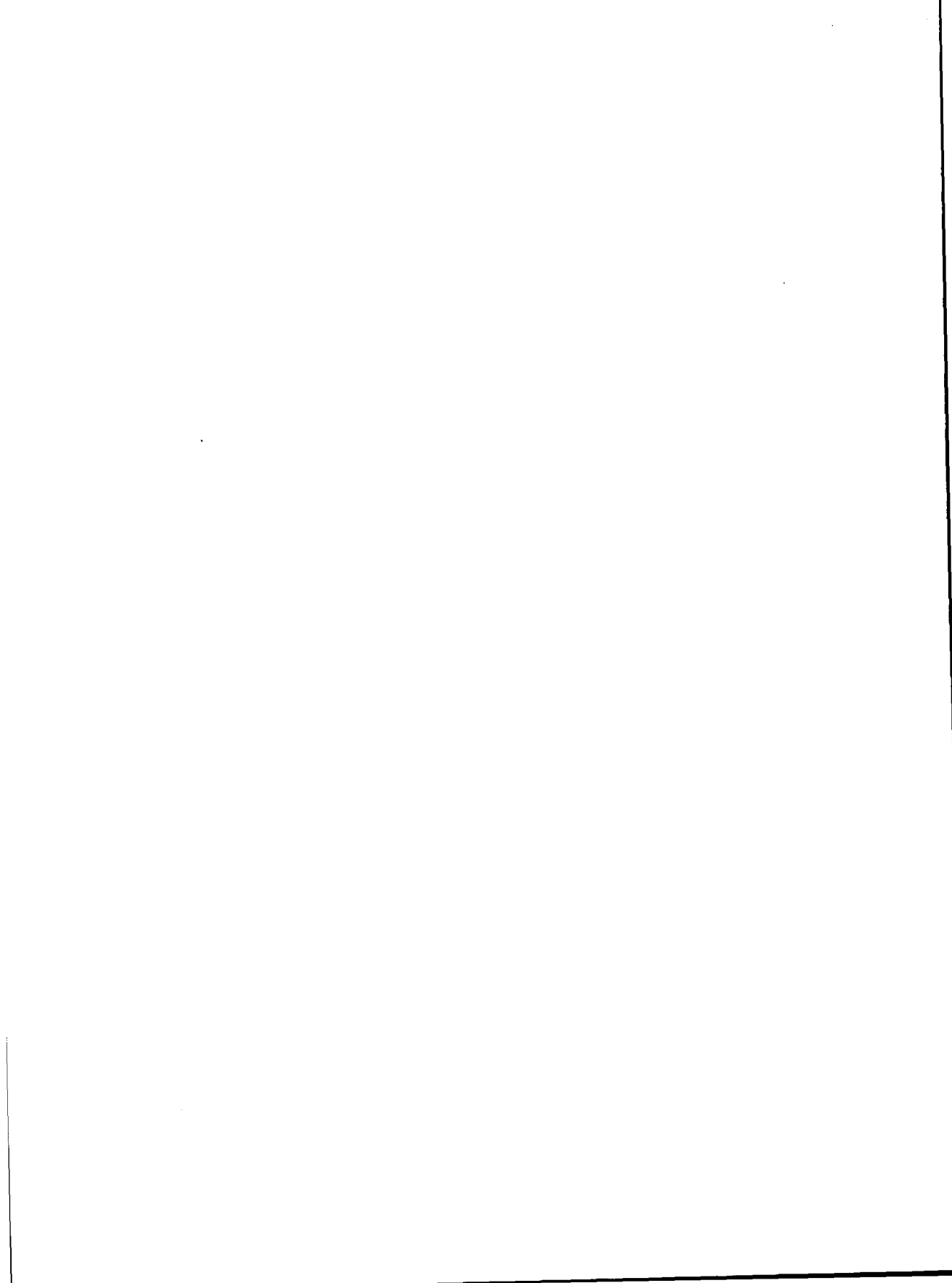
wait attributes  
  for idle threads 315  
wait\_barrier function 187  
wait\_barrier\_8 function 187  
wall clock time  
  defined 374  
-Wl, option 327  
word  
  defined 374  
workload-based dynamic selection 102

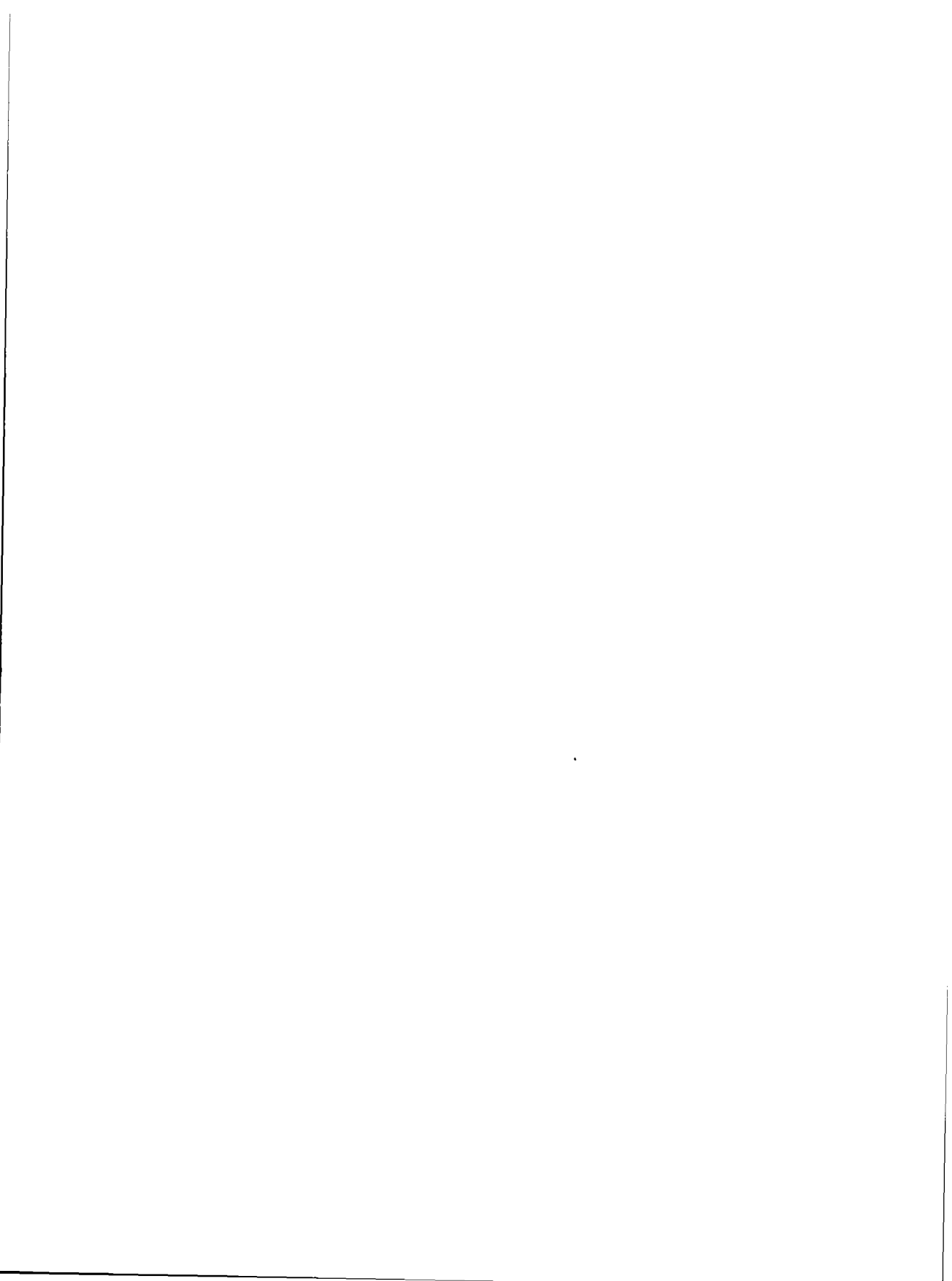
workstation  
  defined 374  
write  
  defined 374  
wrong answers  
  and aliases 230  
  and code motion 230  
  and floating point imprecision 237  
  and large trip counts 253  
  and memory classes 242  
  and no\_loop\_dependence 249

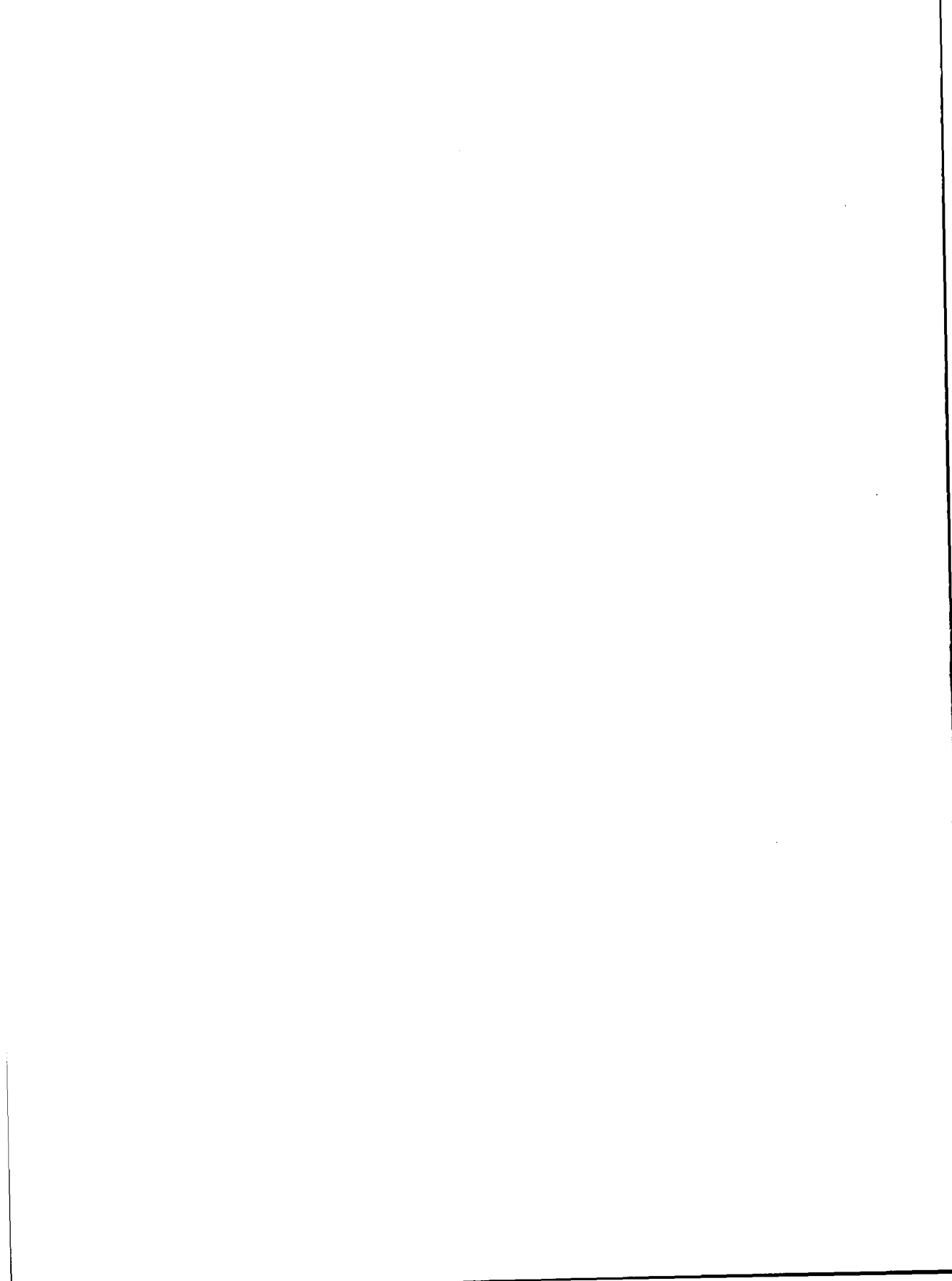
---

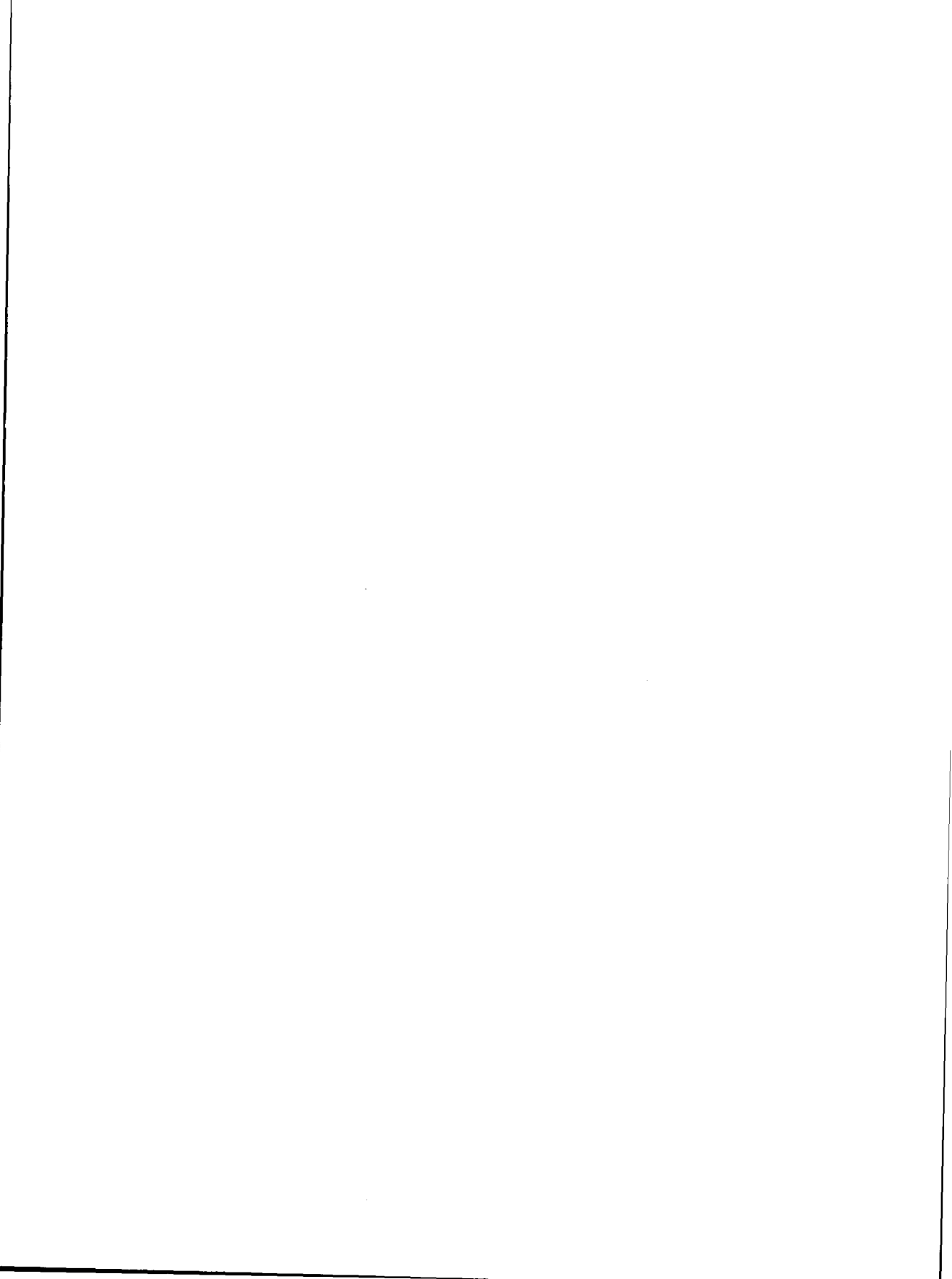
## Z

zero  
  defined 374  
zero stride 251











ORDER NUMBER  
DSW-067

DOCUMENT NUMBER  
710-030230-003

