

# **Parallel Programming Guide for HP-UX Systems**

**K-Class and V-Class Servers**

**Second Edition**



B3909-90003

**Customer Order Number: B3909-90003**

**March 2000**

---

## Revision History

**Edition:** Second

**Document Number:** B3909-90003

**Remarks:** March 2000. Added OpenMP appendix.

**Edition:** First

**Document Number:** B6056-96006

**Remarks:** June 1998. Initial Release.

## Notice

© Copyright Hewlett-Packard Company 2000. All Rights Reserved.  
Reproduction, adaptation, or translation without prior written  
permission is prohibited, except as allowed under the copyright laws.

The information contained in this document is subject to change without  
notice.

Hewlett-Packard makes no warranty of any kind with regard to this  
material, including, but not limited to, the implied warranties of  
merchantability and fitness for a particular purpose. Hewlett-Packard  
shall not be liable for errors contained herein or for incidental or  
consequential damages in connection with the furnishing, performance  
or use of this material.

---

---

# Contents

<b>Preface</b> .....	<b>xv</b>
Scope .....	xvi
Notational conventions .....	xvii
Command syntax .....	xviii
Associated documents .....	xix
<b>1 Introduction</b> .....	<b>1</b>
HP SMP architectures .....	2
Bus-based systems .....	2
Hyperplane Interconnect systems .....	2
Parallel programming model .....	3
The shared-memory paradigm .....	3
The message-passing paradigm .....	4
Overview of HP optimizations .....	6
Basic scalar optimizations .....	6
Advanced scalar optimizations .....	7
Parallelization .....	7
<b>2 Architecture overview</b> .....	<b>9</b>
System architectures .....	10
Data caches .....	12
Data alignment .....	12
Cache thrashing .....	13
Memory Systems .....	17
Physical memory .....	17
Virtual memory .....	17
Interleaving .....	18
Variable-sized pages on HP-UX .....	23
Specifying a page size .....	23
<b>3 Optimization levels</b> .....	<b>25</b>
HP optimization levels and features .....	26
Cumulative Options .....	29
Using the Optimizer .....	30
General guidelines .....	30
C and C++ guidelines .....	31
Fortran guidelines .....	33
<b>4 Standard optimization features</b> .....	<b>35</b>

---

Machine instruction level optimizations (+00) . . . . .	36
Constant folding . . . . .	36
Partial evaluation of test conditions . . . . .	36
Simple register assignment . . . . .	37
Data alignment on natural boundaries . . . . .	37
Block level optimizations (+01) . . . . .	39
Branch optimization . . . . .	39
Dead code elimination . . . . .	40
Faster register allocation . . . . .	40
Instruction scheduling . . . . .	41
Peephole optimizations . . . . .	41
Routine level optimizations (+02) . . . . .	42
Advanced constant folding and propagation . . . . .	42
Common subexpression elimination . . . . .	43
Global register allocation (GRA) . . . . .	43
Register allocation in C and C++ . . . . .	44
Loop-invariant code motion . . . . .	45
Loop unrolling . . . . .	45
Register reassociation . . . . .	47
Software pipelining . . . . .	49
Prerequisites of pipelining . . . . .	49
Strength reduction of induction variables and constants . . . . .	51
Store and copy optimization . . . . .	52
Unused definition elimination . . . . .	52
<b>5 Loop and cross-module optimization features . . . . .</b>	<b>53</b>
Strip mining . . . . .	54
Inlining within a single source file . . . . .	55
Cloning within a single source file . . . . .	57
Data localization . . . . .	58
Conditions that inhibit data localization . . . . .	59
Loop-carried dependences (LCDs) . . . . .	59
Other loop fusion dependences . . . . .	64
Aliasing . . . . .	64
Computed or assigned GOTO statements in Fortran . . . . .	67
I/O statements . . . . .	67
Multiple loop entries or exits . . . . .	68
RETURN or STOP statements in Fortran . . . . .	68
return or exit statements in C or C++ . . . . .	68
throw statements in C++ . . . . .	69
Procedure calls . . . . .	69
Loop blocking . . . . .	70

---

Data reuse . . . . .	71
Spatial reuse . . . . .	71
Temporal reuse . . . . .	72
Loop distribution . . . . .	77
Loop fusion . . . . .	79
Loop interchange . . . . .	82
Loop unroll and jam . . . . .	84
Preventing loop reordering . . . . .	89
Test promotion . . . . .	90
Cross-module cloning . . . . .	91
Global and static variable optimizations . . . . .	91
Global variable optimization coding standards . . . . .	91
Inlining across multiple source files . . . . .	92
<b>6 Parallel optimization features . . . . .</b>	<b>93</b>
Levels of parallelism . . . . .	94
Loop-level parallelism . . . . .	94
Automatic parallelization . . . . .	95
Threads . . . . .	96
Loop transformations . . . . .	97
Idle thread states . . . . .	100
Determining idle thread states . . . . .	100
Parallel optimizations . . . . .	102
Dynamic selection . . . . .	102
Workload-based dynamic selection . . . . .	102
dynsel, no_dynsel . . . . .	103
Inhibiting parallelization . . . . .	105
Loop-carried dependences (LCDs) . . . . .	105
Reductions . . . . .	108
Preventing parallelization . . . . .	110
Parallelism in the aC++ compiler . . . . .	111
Cloning across multiple source files . . . . .	112
<b>7 Controlling optimization . . . . .</b>	<b>113</b>
Command-line optimization options . . . . .	114
Invoking command-line options . . . . .	117
+O[no]aggressive . . . . .	117
+O[no]all . . . . .	118
+O[no]autopar . . . . .	118
+O[no]conservative . . . . .	119
+O[no]dataprefetch . . . . .	119

---

+O[no]dynsel .....	120
+O[no]entrysched.....	120
+O[no]fail_safe.....	121
+O[no]fastaccess.....	121
+O[no]fltacc .....	121
+O[no]global_ptrs_unique[= <i>namelist</i> ] .....	122
+O[no]info .....	123
+O[no]initcheck.....	123
+O[no]inline[= <i>namelist</i> ].....	124
+Oinline_budget= <i>n</i> .....	125
+O[no]libcalls .....	125
+O[no]limit .....	126
+O[no]loop_block.....	127
+O[no]loop_transform .....	127
+O[no]loop_unroll[= <i>unroll factor</i> ] .....	127
+O[no]loop_unroll_jam .....	128
+O[no]moveflops .....	128
+O[no]multiprocessor .....	129
+O[no]parallel .....	129
+O[no]parmsoverlap.....	130
+O[no]pipeline .....	130
+O[no]procelim.....	131
+O[no]ptrs_ansi.....	131
+O[no]ptrs_strongly_typed .....	132
+O[no]ptrs_to_globals[= <i>namelist</i> ] .....	135
+O[no]regreassoc.....	136
+O[no]report[= <i>report_type</i> ].....	137
+O[no]sharedgra.....	138
+O[no]signedpointers .....	138
+O[no]size .....	138
+O[no]static_prediction .....	139
+O[no]vectorize.....	139
+O[no]volatile.....	140
+O[no]whole_program_mode .....	140
+tm <i>target</i> .....	141
<b>C aliasing options.....</b>	<b>143</b>
<b>Optimization directives and pragmas .....</b>	<b>146</b>
<b>Rules for usage .....</b>	<b>147</b>
block_loop[ (block_factor= <i>n</i> ) ].....	148
dynsel[ ( <i>trip_count</i> = <i>n</i> ) ].....	148
no_block_loop .....	148
no_distribute .....	148
no_dynsel .....	149
no_loop_dependence( <i>namelist</i> ) .....	149

---

no_loop_transform .....	149
no_parallel.....	149
no_side_effects( <i>funclist</i> ) .....	150
unroll_and_jam[(unroll_factor= <i>n</i> )] .....	150
<b>8 Optimization Report .....</b>	<b>151</b>
Optimization Report contents.....	152
Loop Report .....	153
Supplemental tables.....	158
Analysis Table .....	158
Privatization Table .....	159
Variable Name Footnote Table .....	160
<b>9 Parallel programming techniques .....</b>	<b>175</b>
Parallelizing directives and pragmas .....	176
Parallelizing loops .....	178
prefer_parallel.....	178
loop_parallel.....	179
Parallelizing loops with calls .....	179
Unparallelizable loops.....	180
prefer_parallel, loop_parallel attributes.....	181
Combining the attributes.....	184
Comparing prefer_parallel, loop_parallel.....	184
Stride-based parallelism .....	186
critical_section, end_critical_section.....	189
Disabling automatic loop thread-parallelization .....	191
Parallelizing tasks .....	192
Parallelizing regions .....	197
Reentrant compilation.....	201
Setting thread default stack size .....	202
Modifying thread stack size.....	202
Collecting parallel information.....	203
Number of processors.....	203
Number of threads .....	204
Thread ID .....	205
Stack memory type.....	205
<b>10 Data privatization .....</b>	<b>207</b>
Directives and pragmas for data privatization .....	208
Privatizing loop variables .....	210
loop_private.....	210
Denoting induction variables in parallel loops .....	212
Secondary induction variables .....	213

---

save_last[ ( <i>list</i> ) ] .....	216
Privatizing task variables .....	218
task_private .....	218
Privatizing region variables .....	220
parallel_private .....	220
Induction variables in region privatization .....	222
<b>11 Memory classes .....</b>	<b>223</b>
Porting multinode applications to single-node servers .....	224
Private versus shared memory .....	225
thread_private .....	225
node_private .....	225
Memory class assignments .....	226
C and C++ data objects .....	227
Static assignments .....	228
thread_private .....	228
node_private .....	231
<b>12 Parallel synchronization .....</b>	<b>233</b>
Thread-parallelism .....	234
Thread ID assignments .....	234
Synchronization tools .....	235
Using gates and barriers .....	235
In C and C++ .....	236
In Fortran .....	236
Synchronization functions .....	237
Allocation functions .....	237
Deallocation functions .....	238
Locking functions .....	239
Unlocking functions .....	240
Wait functions .....	241
sync_routine .....	242
loop_parallel(ordered) .....	245
Critical sections .....	247
Ordered sections .....	248
Synchronizing code .....	250
Using critical sections .....	250
Using ordered sections .....	253
Manual synchronization .....	258
<b>13 Troubleshooting .....</b>	<b>265</b>
Aliasing .....	266
ANSI algorithm .....	266



---

Type-safe algorithm . . . . .	266
Specifying aliasing modes . . . . .	267
Iteration and stop values . . . . .	267
Using potential aliases as addresses of variables . . . . .	267
Using hidden aliases as pointers . . . . .	268
Using a pointer as a loop counter. . . . .	268
Aliasing stop variables . . . . .	269
Global variables . . . . .	269
False cache line sharing . . . . .	271
Aligning data to avoid false sharing . . . . .	274
Aligning arrays on cache line boundaries . . . . .	274
Distributing iterations on cache line boundaries . . . . .	275
Thread-specific array elements . . . . .	276
Scalars sharing a cache line. . . . .	277
Working with unaligned arrays. . . . .	278
Working with dependences . . . . .	279
Floating-point imprecision . . . . .	281
Enabling sudden underflow. . . . .	282
Invalid subscripts . . . . .	283
Misused directives and pragmas . . . . .	284
Loop-carried dependences . . . . .	284
Reductions. . . . .	286
Nondeterminism of parallel execution . . . . .	287
Triangular loops. . . . .	288
Parallelizing the outer loop . . . . .	290
Parallelizing the inner loop . . . . .	290
Examining the code . . . . .	294
Compiler assumptions. . . . .	296
Incrementing by zero . . . . .	296
Trip counts that may overflow. . . . .	297
Linear test replacement . . . . .	297
Large trip counts at +02 and above . . . . .	299

**Appendix A: Porting CPSlib functions to pthreads301**

Introduction . . . . .	301
Accessing pthreads. . . . .	302
Mapping CPSlib functions to pthreads. . . . .	303
Environment variables . . . . .	309
Using pthreads. . . . .	310
Symmetric parallelism. . . . .	310
ppcall.c . . . . .	311

---

Asymmetric parallelism .....	321
create.c .....	322
pth_create.c .....	323
Synchronization using high-level functions .....	324
Barriers .....	324
Mutexes .....	327
Synchronization using low-level functions .....	329
Low-level locks .....	329
Low-level counter semaphores .....	329
<b>Appendix B: OpenMP Parallel Programming Model.....</b>	<b>333</b>
What is OpenMP? .....	334
HP's Implementation of OpenMP .....	335
OpenMP Command-line Options .....	335
Default .....	335
OpenMP Directives .....	335
OpenMP Data Scope Clauses .....	336
Other Supported OpenMP Clauses .....	336
From HP Programming Model to OpenMP .....	337
Syntax .....	337
Exceptions .....	338
HP Programming Model Directives .....	339
Not Accepted with +Openmp .....	339
Accepted with +Openmp .....	340
More Information on OpenMP .....	341

---

## Figures

Figure 1	Symmetric multiprocessor system . . . . .	3
Figure 2	Message-passing programming model . . . . .	4
Figure 3	K-Class bus configuration . . . . .	10
Figure 4	V2250 Hyperplane Interconnect view . . . . .	11
Figure 5	Array layouts—cache-thrashing . . . . .	14
Figure 6	Array layouts—non-thrashing . . . . .	15
Figure 7	V2250 interleaving . . . . .	19
Figure 8	V2250 interleaving of arrays A and B . . . . .	22
Figure 9	LCDs in original and interchanged loops . . . . .	62
Figure 10	Values read into array A . . . . .	67
Figure 11	Blocked array access . . . . .	73
Figure 12	Spatial reuse of A and B . . . . .	74
Figure 13	One-dimensional parallelism in threads . . . . .	97
Figure 14	Conceptual strip mine for parallelization . . . . .	98
Figure 15	Parallelized loop . . . . .	99
Figure 16	Stride-parallelized loop . . . . .	187
Figure 17	Ordered parallelization . . . . .	246
Figure 18	LOOP_PARALLEL(ORDERED) synchronization . . . . .	254
Figure 19	Data ownership by CHUNK and NTCHUNK blocks . . . . .	293

---

---

## Tables

Table 1	Locations of HP compilers . . . . .	.25
Table 2	Optimization levels and features . . . . .	.26
Table 3	Loop transformations affecting data localization . . . . .	.58
Table 4	Form of <code>no_loop_dependence</code> directive and pragma . . . . .	.60
Table 5	Computation sequence of $A(I, J)$ : original loop . . . . .	.61
Table 6	Computation sequence of $A(I, J)$ : interchanged loop . . . . .	.62
Table 7	Forms of <code>block_loop</code> , <code>no_block_loop</code> directives and pragmas . . . . .	.70
Table 8	Form of <code>no_distribute</code> directive and pragma . . . . .	.77
Table 9	Forms of <code>unroll_and_jam</code> , <code>no_unroll_and_jam</code> directives and pragmas . . . . .	.85
Table 10	Form of <code>no_loop_transform</code> directive and pragma . . . . .	.89
Table 11	Form of <code>MP_IDLE_THREADS_WAIT</code> environment variable . . . . .	.100
Table 12	Form of <code>dynsel</code> directive and pragma . . . . .	.103
Table 13	Form of <code>reduction</code> directive and pragma . . . . .	.108
Table 14	Form of <code>no_parallel</code> directive and pragma . . . . .	.110
Table 15	Command-line optimization options . . . . .	.114
Table 16	<code>+O[no]fltacc</code> and floating-point optimizations . . . . .	.122
Table 17	Optimization Report contents . . . . .	.137
Table 18	<code>+tm target</code> and <code>+DA/+DS</code> . . . . .	.142
Table 19	Directive-based optimization options . . . . .	.146
Table 20	Form of optimization directives and pragmas . . . . .	.147
Table 21	Optimization Report contents . . . . .	.152
Table 22	Loop Report column definitions . . . . .	.154
Table 23	Reordering transformation values in the Loop Report . . . . .	.155
Table 24	Optimizing/special transformations values in the Loop Report . . . . .	.157
Table 25	Analysis Table column definitions . . . . .	.158
Table 26	Privatization Table column definitions . . . . .	.159
Table 27	Variable Name Footnote Table column definitions . . . . .	.160
Table 28	Parallel directives and pragmas . . . . .	.176
Table 29	Forms of <code>prefer_parallel</code> and <code>loop_parallel</code> directives and pragmas . . . . .	.181
Table 30	Attributes for <code>loop_parallel</code> , <code>prefer_parallel</code> . . . . .	.182
Table 31	Comparison of <code>loop_parallel</code> and <code>prefer_parallel</code> . . . . .	.185
Table 32	Iteration distribution using <code>chunk_size = 1</code> . . . . .	.186
Table 33	Iteration distribution using <code>chunk_size = 5</code> . . . . .	.186
Table 34	Forms of <code>critical_section/end_critical_section</code> directives and pragmas . . . . .	.189
Table 35	Forms of task parallelization directives and pragmas . . . . .	.192
Table 36	Attributes for task parallelization . . . . .	.193
Table 37	Forms of region parallelization directives and pragmas . . . . .	.198
Table 38	Attributes for region parallelization . . . . .	.198

---

Table 39	Forms of <code>CPS_STACK_SIZE</code> environment variable	202
Table 40	Number of processors functions	204
Table 41	Number of threads functions	204
Table 42	Thread ID functions	205
Table 43	Stack memory type functions	205
Table 44	Data Privatization Directives and Pragmas	208
Table 45	Form of <code>loop_private</code> directive and pragma	210
Table 46	Form of <code>save_last</code> directive and pragma	216
Table 47	Form of <code>task_private</code> directive and pragma	218
Table 48	Form of <code>parallel_private</code> directive and pragma	220
Table 49	Form of memory class directives and variable declarations	226
Table 50	Forms of gate and barriers variable declarations	235
Table 51	Forms of allocation functions	237
Table 52	Forms of deallocation functions	238
Table 53	Forms of locking functions	239
Table 54	Form of unlocking functions	240
Table 55	Form of wait functions	241
Table 56	Form of <code>sync_routine</code> directive and pragma	242
Table 57	Forms of <code>critical_section</code> , <code>end_critical_section</code> directives and pragmas	247
Table 58	Forms of <code>ordered_section</code> , <code>end_ordered_section</code> directives and pragmas	248
Table 59	Initial mapping of array to cache lines	272
Table 60	Default distribution of the <code>I</code> loop	273
Table 61	CPSlib library functions to pthreads mapping	303
Table 62	CPSlib environment variables	309
Table 63	OpenMP Directives and Required Opt Levels	335
Table 64	OpenMP and HPPM Directives/Clauses	337

---

# Preface

This guide describes efficient methods for shared-memory programming using the following HP-UX compilers: HP Fortran 90, HP aC++ (ANSI C++), and HP C.

The *Parallel Programming Guide for HP-UX* is intended for use by experienced Fortran 90, C, and C++ programmers. This guide describes the enhanced features of HP-UX 11.0 compilers on single-node multiprocessor HP technical servers. These enhancements include new loop optimizations and constructs for creating programs to run concurrently on multiple processors.

You need not be familiar with the HP parallel architecture, programming models, or optimization concepts to understand the concepts introduced in this book.

---

## Scope

This guide covers programming methods for the following HP compilers on V2200 and V2250 and K-Class machines running HP-UX 11.0 and higher:

- HP Fortran 90 Version 2.0 (and higher)
- HP aC++ Version 3.0 (and higher)
- HP C Version 1.2.3 (and higher)

The HP compilers now support an extensive shared-memory programming model. HP-UX 11.0 and higher includes the required assembler, linker, and libraries.

This guide describes how to produce programs that efficiently exploit the features of HP parallel architecture concepts and the HP compiler set. Producing efficient programs requires the use of efficient algorithms and implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. It is assumed that you have chosen the best possible algorithm for your problem. This manual should help you obtain the best possible performance from that algorithm.



---

## Notational conventions

This section discusses notational conventions used in this book.

<b>bold monospace</b>	In command examples, <b>bold monospace</b> identifies input that must be typed exactly as shown.
monospace	In paragraph text, <code>monospace</code> identifies command names, system calls, and data structures and types. In command examples, <code>monospace</code> identifies command output, including error messages.
<i>italic</i>	In paragraph text, <i>italic</i> identifies titles of documents. In command syntax diagrams, <i>italic</i> identifies variables that you must provide. The following command example uses brackets to indicate that the variable <i>output_file</i> is optional: <code>command input_file [output_file]</code>
Brackets ( [ ] )	In command examples, square brackets designate optional entries.
Curly brackets ( { } ), Pipe (   )	In command syntax diagrams, text surrounded by curly brackets indicates a choice. The choices available are shown inside the curly brackets and separated by the pipe sign (   ). The following command example indicates that you can enter either a or b: <code>command {a   b}</code>

Horizontal ellipses (...)	In command examples, horizontal ellipses show repetition of the preceding items.
Vertical ellipses	Vertical ellipses show that lines of code have been left out of an example.
<b>Keycap</b>	<b>Keycap</b> indicates the keyboard keys you must press to execute the command example.

The term “Fortran” refers to Fortran 90.

The directives and pragmas described in this book can be used with the Fortran 90 and C compilers, unless otherwise noted. The aC++ compiler does not support the pragmas, but does support the memory classes. In general discussion, these directives and pragmas are presented in lowercase type, but each compiler recognizes them regardless of their case.

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, type:

```
% man 1 mnpname
```

NOTE

A Note highlights important supplemental information.

## Command syntax

Consider this example:

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

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

---

## Associated documents

The following documents are listed as additional resources to help you use the compilers and associated tools:

- *Fortran 90 Programmer's Guide*—Provides extensive usage information (including how to compile and link), suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines for HP Fortran 90.
- *HP Fortran 90 Programmer's Notes*—Provides usage information, including instructions on how to compile and link, suggestions and tools for migrating to HP Fortran 90, and details on calling C and HP-UX routines from HP Fortran 90.
- *Fortran 90 Programmer's Reference*—Presents complete Fortran 90 language reference information. It also covers compiler options, compiler directives, and library information.
- *HP aC++ Online Programmer's Guide*—Presents reference and tutorial information on aC++. This manual is only available in html format.
- *HP MPI User's Guide*—Discusses message-passing programming using Hewlett-Packard's Message-Passing Interface library.
- *Programming with Threads on HP-UX*—Discusses programming with POSIX threads.
- *HP C/HP-UX Reference Manual*—Presents reference information on the C programming language, as implemented by HP.
- *HP C/HP-UX Programmer's Guide*—Contains detailed discussions of selected C topics.
- *HP-UX Linker and Libraries User's Guide*—Describes how to develop software on HP-UX, using the HP compilers, assemblers, linker, libraries, and object files.
- *Managing Systems and Workgroups*—Describes how to perform various system administration tasks.

## Preface

- *Threadtime* by Scott J. Norton and Mark D. DiPasquale—Provides detailed guidelines on the basics of thread management, including POSIX thread structure; thread management functions; and the creation, termination and synchronization of threads.

---

# 1

# Introduction

Hewlett-Packard compilers generate efficient parallel code with little user intervention. However, you can increase this efficiency by using the techniques discussed in this book.

This chapter contains a discussion of the following topics:

- HP SMP architectures
- Parallel programming model
- Overview of HP optimizations

## HP SMP architectures

Hewlett-Packard offers single-processor and symmetric multiprocessor (SMP) systems. This book focuses on SMP systems, specifically, those that utilize different bus configurations for memory access. These are briefly described in the following sections, and in more detail in the “Architecture overview” section on page 9.

### Bus-based systems

The K-Class servers are midrange servers with a bus-based architecture. It contains one set of processors and physical memory. Memory is shared among all the processors, with a bus serving as the interconnect. The shared-memory architecture has a uniform access time from each processor.

### Hyperplane Interconnect systems

The V-Class servers configurations range from one to 16 processors on the V-Class single-node system. These systems have the following characteristics:

- Processors communicate with each other through memory and by using I/O devices through a Hyperplane Interconnect nonblocking crossbar.
- Scalable physical memory. The current V-Class server support up to 16 Gbytes of memory.
- Each process on an HP system can access a 16-terabyte (Tbyte) virtual address space.

---

## Parallel programming model

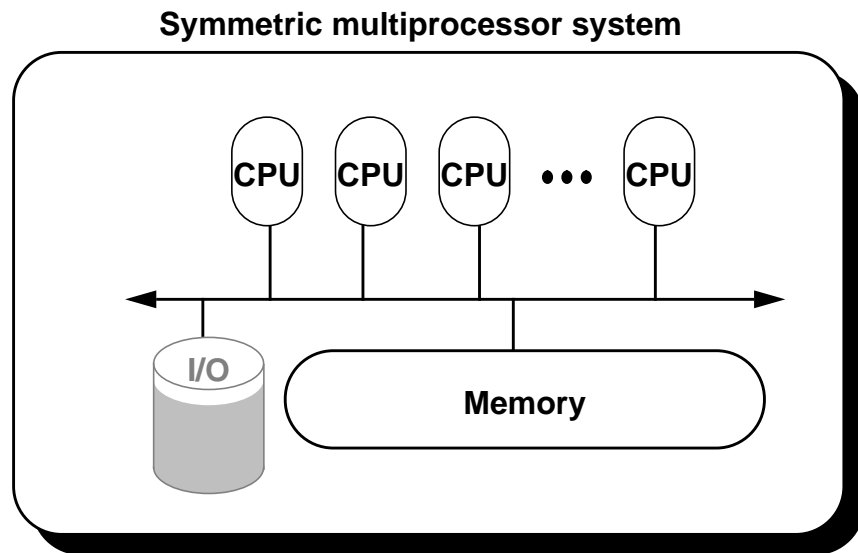
Parallel programming models provide perspectives from which you can write—or adapt—code to run on a high-end HP system. You can perform both shared-memory programming and message-passing programming on an SMP. This book focuses on using the shared-memory paradigm, but includes reference material and pointers to other manuals about message passing.

### The shared-memory paradigm

In the shared-memory paradigm, compilers handle optimizations, and, if requested, parallelization. Numerous compiler directives and pragmas are available to further increase optimization opportunities. Parallelization can also be specified using POSIX threads (Pthreads). Figure 1 shows the SMP model for the shared-memory paradigm.

Figure 1

#### Symmetric multiprocessor system



The directives and pragmas associated with the shared-memory programming model are discussed in “Parallel programming techniques,” on page 175, “Memory classes,” on page 223, and “Parallel synchronization,” on page 233.

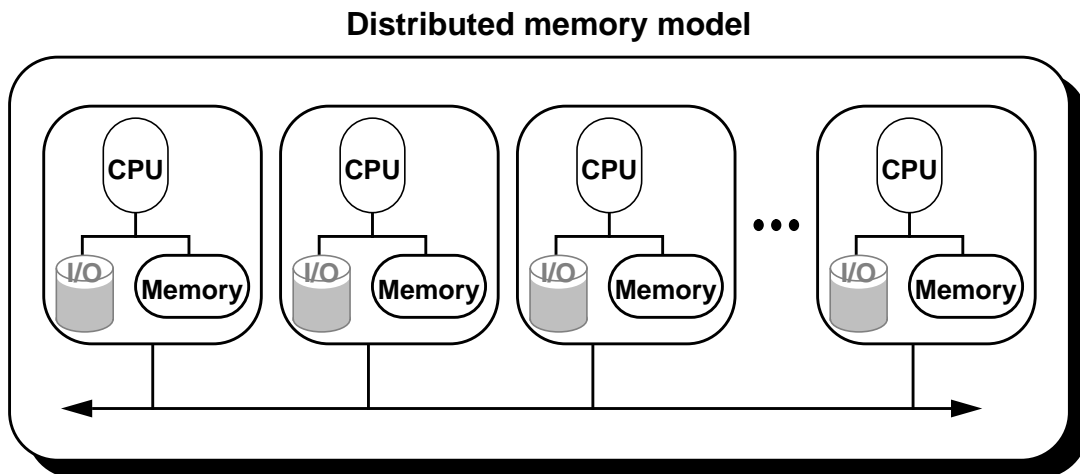
## The message-passing paradigm

HP has implemented a version of the message-passing interface (MPI) standard known as HP MPI. This implementation is finely tuned for HP technical servers.

In message-passing, a parallel application consists of a number of processes that run concurrently. Each process has its own local memory. It communicates with other processes by sending and receiving messages. When data is passed in a message, both processes must work to transfer the data from the local memory of one to the local memory of the other.

Under the message-passing paradigm, functions allow you to explicitly spawn parallel processes, communicate data among them, and coordinate their activities. Unlike the previous model, there is no shared-memory. Each process has its own private 16-terabyte (Tbyte) address space, and any data that must be shared must be explicitly passed between processes. Figure 2 shows a layout of the message-passing paradigm.

**Figure 2** Message-passing programming model





Support of message passing allows programs written under this paradigm for distributed memory to be easily ported to HP servers. Programs that require more per-process memory than possible using shared-memory benefit from the manually-tuned message-passing style. For more information about HP MPI, see the *HP MPI User's Guide* and the *MPI Reference*.

## Overview of HP optimizations

HP compilers perform a range of user-selectable optimizations. These new and standard optimizations, specified using compiler command-line options, are briefly introduced here. A more thorough discussion, including the features associated with each, is provided in “Optimization levels,” on page 25.

### Basic scalar optimizations

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

A basic block is a sequence of statements that has a single entry point and 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 and C++. Program units are also generically referred to as procedures. Basic blocks are contained within program units. Optimizations at the program unit level span basic blocks.

To improve performance, basic optimizations perform the following activities:

- 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 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 memory accesses.

Advanced scalar optimizations include several loop transformations. Many of these optimizations 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 “Controlling optimization,” on page 113.

Advanced scalar optimizations implicitly include all basic scalar optimizations.

## Parallelization

HP compilers automatically locate and exploit loop-level parallelism in most programs. Using the techniques described in Chapter 9, “Parallel programming techniques”, you can help the compilers find even more parallelism in your programs.

Loops that have been data-localized are prime candidates for parallelization. Individual iterations of loops that contain strips of localizable data are parcelled out among several processors and run simultaneously. For example, the maximum number of processors that can be used is limited by the number of iterations of the loop and 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 both basic and advanced scalar optimizations.

Introduction  
Overview of HP optimizations

---

## 2

# Architecture overview

This chapter provides an overview of Hewlett-Packard's shared memory K-Class and V-Class architectures. The information in this chapter focuses on this architecture as it relates to parallel programming.

This chapter describes architectural features of HP's K-Class and V-Class. For more information on the family of V-Class servers, see the *V-Class Architecture* manual.

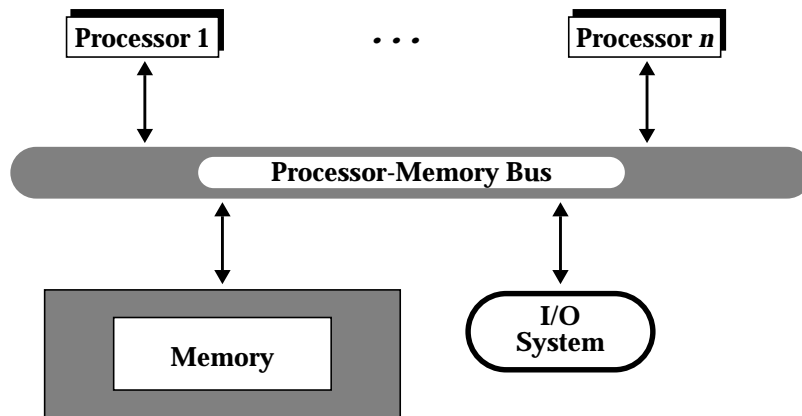
---

## System architectures

PA-RISC processors communicate with each other, with memory, and with peripherals through various bus configuration. The difference between the K-Class and V-Class servers are presented by the manner in which they access memory. The K-Class maintains a bus-based configuration, shown in Figure 3.

**Figure 3**

### K-Class bus configuration

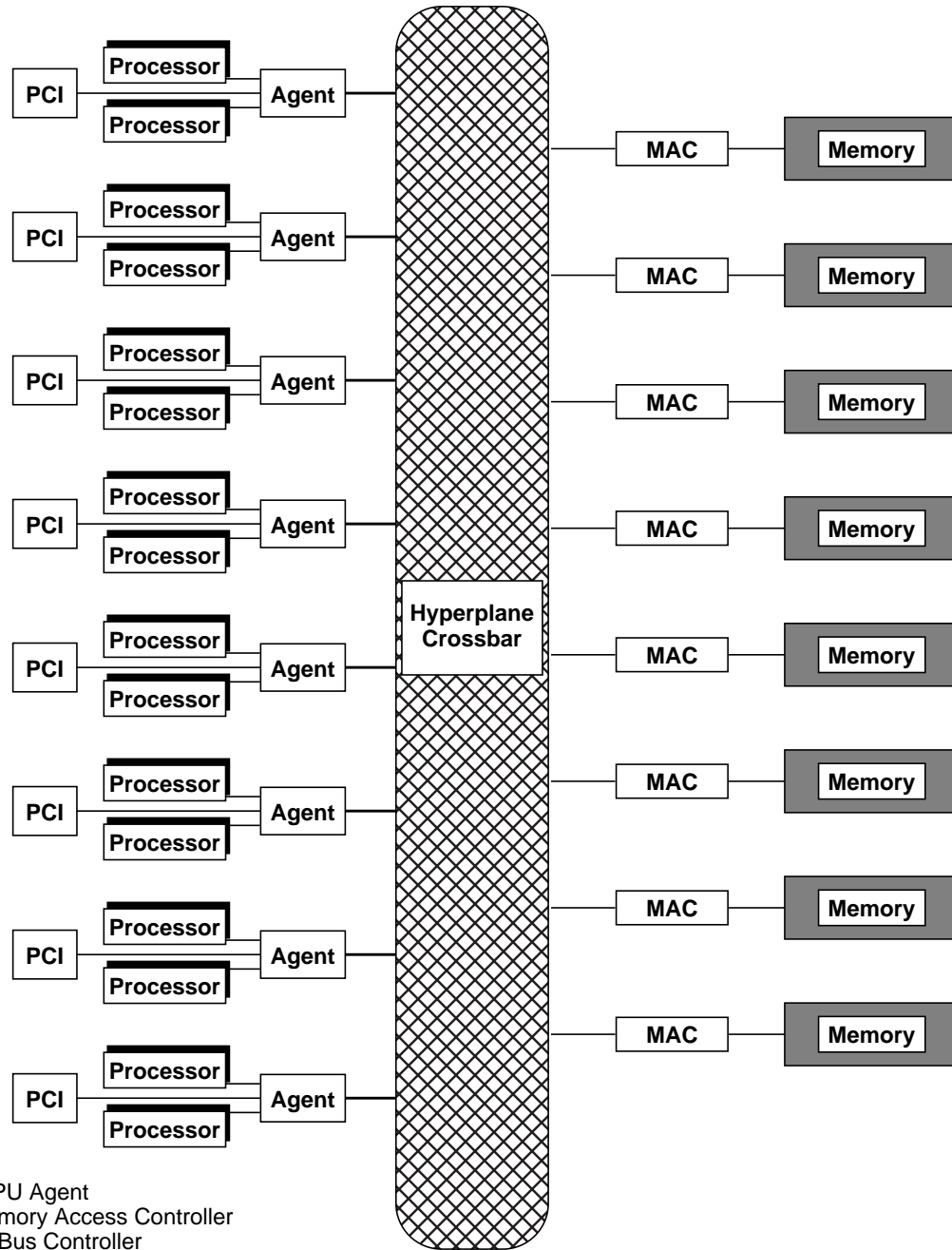


On a V-Class, processors communicate with each other, memory, and peripherals through a nonblocking crossbar. The V-Class implementation is achieved through the Hyperplane Interconnect, shown in Figure 4.

The HP V2250 server has one to 16 PA-8200 processors and 256 Mbytes to 16 Gbytes of physical memory. Two CPUs and a PCI bus share a single CPU agent. The CPUs communicate with the rest of the machine through the CPU agent. The Memory Access Controllers (MACs) provide the interface between the memory banks and the rest of the machine.

CPUs communicate directly with their own instruction and data caches, which are accessed by the processor in one clock (assuming a full pipeline). V2250 servers use 2-Mbyte off-chip instruction caches and data caches.

**Figure 4** V2250 Hyperplane Interconnect view



## Data caches

HP systems use cache to enhance performance. Cache sizes, as well as cache line sizes, vary with the processor used. Data is moved between the cache and memory using cache lines. 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—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 a number of other data objects that were not specifically requested.

One 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 originally requested element are referenced by the program, or if the originally 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, as found on a V2250, invalidates cache lines in other processors when they are stored to by a particular processor. This indicates to other processors that they must load the cache line from memory the next time they reference its data.

## Data alignment

Aligning data addresses on cache line boundaries allows for efficient data reuse in loops (refer to “Data reuse” on page 71). The linker automatically aligns data objects larger than 32 bytes in size on a 32-byte boundary. It also aligns data greater than a page size on a 64-byte boundary.

Only the first item in a list of data objects appearing in any of these statements is aligned on a cache line boundary. To make the most efficient use of available memory, the total size, in bytes, of any array appearing in one of these statements should be an integral multiple of 32.

Sizing your arrays this way prevents data following the first array from becoming misaligned. Scalar variables should be listed after arrays and ordered from longest data type to shortest. For example, `REAL*8` scalars should precede `REAL*4` scalars.



You can align data on 64-byte boundaries by doing the following. These apply only to parallel executables:

- Using Fortran `ALLOCATE` statements
- Using the C functions `malloc` or `memory_class_malloc`

**NOTE**

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

## Cache thrashing

Cache thrashing occurs when two or more data items that are frequently needed by the program both map to the same cache address. Each time one of the items is encached, it overwrites another needed item, causing cache misses and impairing data reuse. This section explains how thrashing happens on the V-Class.

A type of thrashing known as false cache line sharing is discussed in the section “False cache line sharing” on page 271.

**Example**

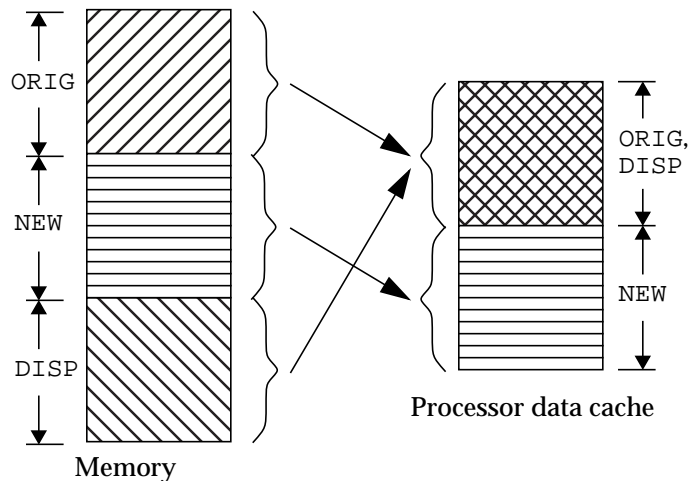
### Cache thrashing

The following Fortran example provides an example of cache thrashing:

```
REAL*8 ORIG(131072), NEW(131072), DISP(131072)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
DO I = 1, N
    NEW(I) = ORIG(I) + DISP(I)
ENDDO
```

In this example, the arrays `ORIG` and `DISP` overwrite each other in a 2-Mbyte cache. Because the arrays are in a `COMMON` block, they are allocated in contiguous memory in the order shown. Each array element occupies 8 bytes, so each array occupies one Mbyte ( $8 \times 131072 = 1048576$  bytes). Therefore, arrays `ORIG` and `DISP` are exactly 2-Mbytes 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 5.

**Figure 5** 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 memory into the cache. Because these elements have identical cache addresses, whichever is fetched last overwrites the first. 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 (three extra `REAL*8` elements are fetched in this case) must remain encached until they are 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.

Memory accesses take substantially longer than cache accesses, which 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.

The problem is easily fixed by increasing the distance between the arrays. You can accomplish this by either increasing the array sizes or inserting a padding array.

**Example**

**Cache padding**

The following Fortran example illustrates cache padding:

```
REAL*8 ORIG(131072), NEW(131072), P(4),DISP(131072)
COMMON /BLK1/ ORIG, NEW, P, DISP
.
.
.
```

In this example, the array P(4) moves DISP 32 bytes further from ORIG in memory. No two elements of the same index share a cache address. This postpones cache overwriting for the given loop until the entire current cache line is completely exploited.

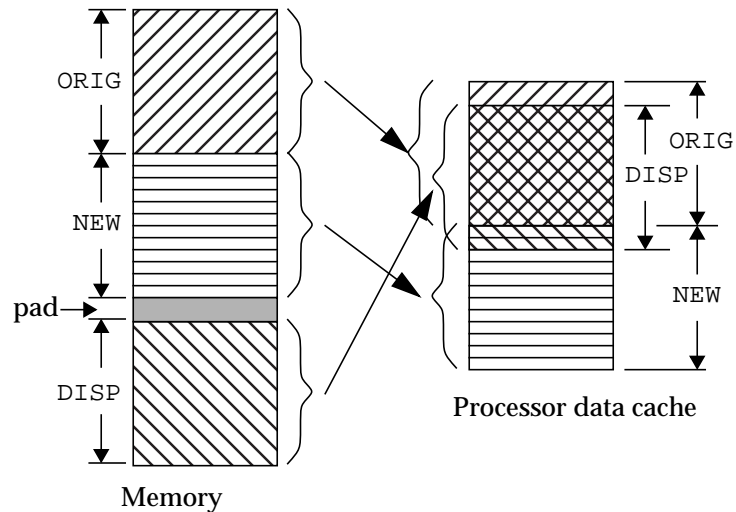
The alternate approach involves increasing the size of ORIG or NEW by 4 elements (32 bytes), as shown in the following example:

```
REAL*8 ORIG(131072), NEW(131080), DISP(131072)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
```

Here, NEW has been increased by 4 elements, providing the padding necessary to prevent ORIG from sharing cache addresses with DISP. Figure 6 shows how both solutions prevent thrashing.

**Figure 6**

**Array layouts—non-thrashing**



[Architecture overview](#)  
[System architectures](#)

It is important to note that this is a highly simplified, worst-case example.

Loop blocking optimization (described in “Loop blocking” on page 70) eliminates thrashing from certain nested loops, but not from all loops. Declaring arrays with dimensions that are not powers of two can help, but it does not completely eliminate the problem.

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.

## Memory Systems

HP's K-Class and V-Class servers maintain a single level of memory latency. Memory functions and interleaving work similarly on both servers, as described in the following sections.

### Physical memory

Multiple, independently accessible memory banks are available on both the K-Class and V-Class servers. In 16-processor V2250 servers, for example, each node consists of up to 32 memory banks. This memory is typically partitioned (by the system administrator) into system-global, and buffer cache. It is also interleaved as described in "Interleaving" on page 18". The K-Class architecture supports up to four memory banks.

System-global memory is accessible by all processors in a given system. The buffer cache is a file system cache and is used to encache items that have been read from disk and items that are to be written to disk.

Memory interleaving is used to improve performance. For an explanation, see the section "Interleaving" on page 18.

### Virtual memory

Each process running on a V-Class or K-Class server under HP-UX accesses its own 16-Tbyte virtual address space. Almost all of this space is available to hold program text, data, and the stack. The space used by the operating system is negligible.

The memory stack size is configurable. Refer to the section "Setting thread default stack size" on page 202 for more information.

Both servers share data among all threads unless a variable is declared to be thread private. Memory class definitions describing data disposition across hypernodes have been retained for the V-Class. This is primarily for potential use when porting to multinode machines.

`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. These addresses map to unique physical addresses in hypernode-local physical memory.

`node_private`

This memory is shared among the threads of a process running on a single node. Since the V-Class and K-Class servers are single-node machines, `node_private` actually serves as one common shared memory class.

Memory classes are discussed more fully in “Memory classes,” on page 223.

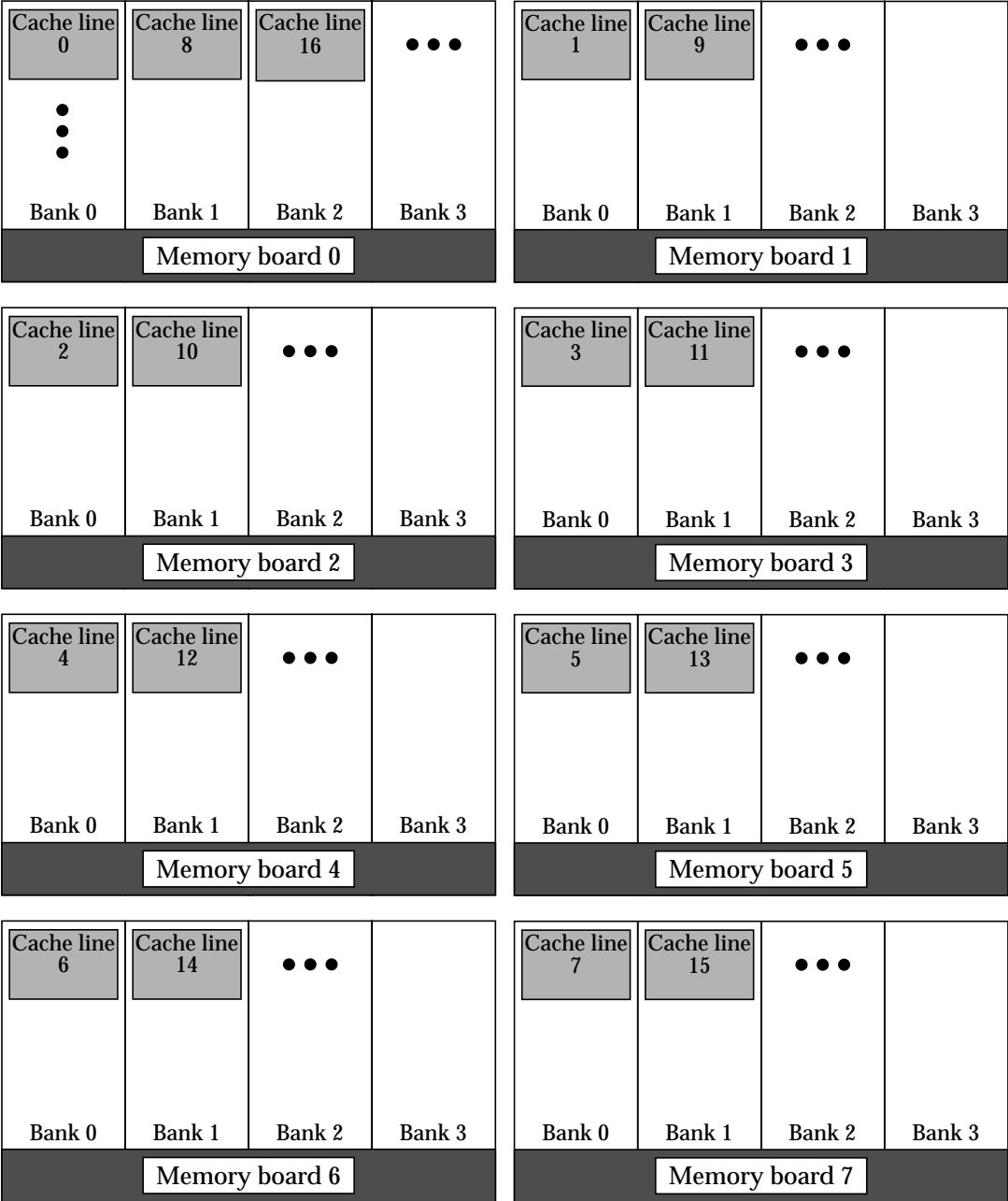
Processes cannot access each other’s virtual address spaces. This virtual memory maps to the physical memory of the system on which the process is running.

## **Interleaving**

Physical pages are interleaved across the memory banks on a cache-line basis. There are up to 32 banks in the V2250 servers; there are up to four on a K-Class. Contiguous cache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown in Figure 7 for V2250 servers.

Interleaving speeds memory accesses by allowing several processors to access contiguous data simultaneously. It also eliminates busy bank and board waits for unit stride accesses. This is beneficial when a loop that manipulates arrays is split among many processors. In the best case, threads access data in patterns with no bank contention. Even in the worst case, in which each thread initially needs the same data from the same bank, after the initial contention delay, the accesses are spread out among the banks.

**Figure 7 V2250 interleaving**



## Example

### Interleaving

The following Fortran 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 memory, with A starting in bank 0, processor cache line 0 for V2250 servers, as shown in Figure 8 on page 22.

You may assume that the HP Fortran 90 compiler parallelizes the J loop to run on as many processors as are available in the system (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. These four processors are identified as CPU0 through CPU3.

#### NOTE

This example is designed to simplify illustration. In reality, the dynamic selection optimization (discussed in “Dynamic selection” on page 102) 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 attempt to simultaneously fetch its portion of the arrays.

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.

Because of the number of memory banks in the V2250 architecture, interleaving removes the contention from the beginning of the loop from the example, as shown in Figure 8.

- 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)



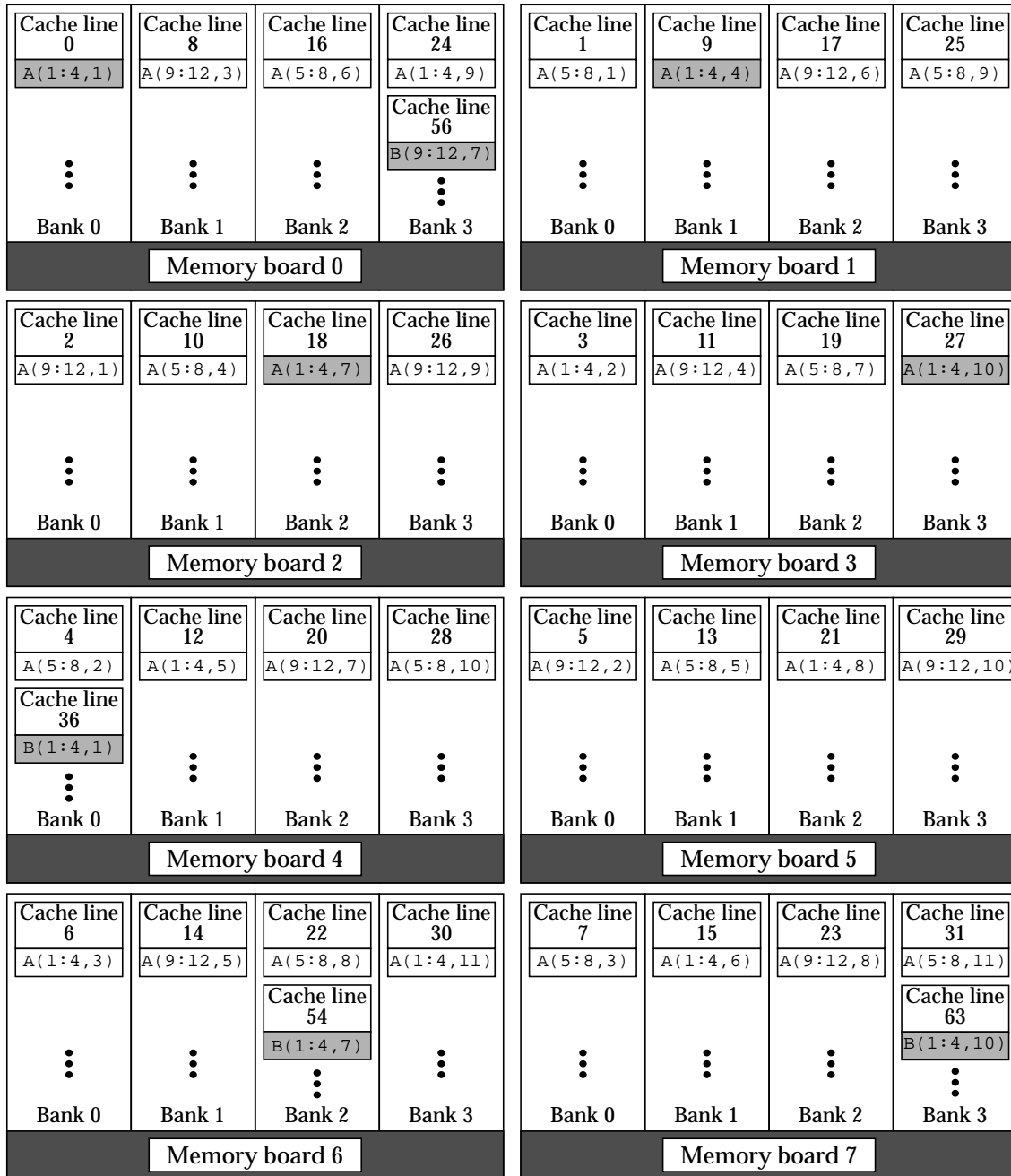
- CPU3 needs  $A(1:12, 10:12)$  and  $B(1:12, 10:12)$

The data from the V2250 example above is spread out on different memory banks as described below:

- $A(1, 1)$ , the first element of the chunk needed by CPU0, is on cache line 0 in bank 0 on board 0
- $A(1, 4)$ , the first element needed by CPU1, is on cache line 9 in bank 1 on board 1
- $A(1, 7)$ , the first element needed by CPU2, is on cache line 18 in bank 2 on board 2
- $A(1, 10)$  the first element needed by CPU3, is on cache line 27 in bank 3 on board 3

Because of interleaving, no contention exists between the processors when trying to read their respective portions of the arrays. Contention may surface occasionally as the processors make their way through the data, but the resulting delays are minimal compared to what could be expected without interleaving.

**Figure 8** V2250 interleaving of arrays A and B



## Variable-sized pages on HP-UX

Variable-sized pages are used to reduce Translation Lookaside Buffer (TLB) misses, improving performance. A TLB is a hardware entity used to hold a virtual to physical address translation. With variable-sized pages, each TLB entry used can map a larger portion of an application's virtual address space. Thus, applications with large data sets are mapped using fewer TLB entries, resulting in fewer TLB misses.

Using a different page size does not help if an application is not experiencing performance degradation due to TLB misses. Additionally, if an application uses too large a page size, fewer pages are available to other applications on the system. This potentially results in increased paging activity and performance degradation.

Valid page sizes on the PA-8200 processors are 4K, 16K, 64K, 256K, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, and 256 Mbytes. The default configurable page size is 4K. Methods for specifying a page size are described below. Note that the user-specified page size only requests a specific size. The operating system takes various factors into account when selecting the page size.

### Specifying a page size

The following `chatr` utility command options allow you to specify information regarding page sizes.

- `+pi` affects the page size for the application's text segment
- `+pd` affects the page size for the application's data segment

The following configurable kernel parameters allow you to specify information regarding page sizes.

- `vps_pagesize` represents the default or minimum page size (in kilobytes) if the user has not used `chatr` to specify a value. The default is 4Kbytes.
- `vps_ceiling` represents the maximum page size (in kilobytes) if the user has not used `chatr` to specify a value. The default is 16Kbytes.
- `vps_chatr_ceiling` places a restriction on the largest value (in kilobytes) a user can specify using `chatr`. The default is 64 Mbytes.

For more information on the `chatr` utility, see the `chatr(1)` man page.

Architecture overview  
**Memory Systems**

---

## 3 Optimization levels

This chapter discusses various optimization levels available with the HP compilers. This contains a discussion of the following topics:

- HP optimization levels and features
- Using the Optimizer

The locations of the compilers discussed in this manual are provided in Table 1.

**Table 1**      **Locations of HP compilers**

<b>Compiler</b>	<b>Description</b>	<b>Location</b>
f90	Fortran 90	/opt/fortran90/bin/f90
cc	ANSI C	/opt/ansic/bin/c89
aC++	ANSI C++	/opt/aCC/bin/aCC

For detailed information about optimization command-line options, and pragmas and directives, see “Controlling optimization,” on page 113.

---

## HP optimization levels and features

This section provides an overview of optimization features which can be through either the command-line optimization options or manual specification using pragmas or directives.

Five optimization levels are available for use with the HP compiler: +00 (the default), +01, +02, +03, and +04. These options have identical names and perform identical optimizations, regardless of which compiler you are using. They can also be specified on the compiler command line in conjunction with other options you may want to use. HP compiler optimization levels are described in Table 2.

**Table 2** Optimization levels and features

Optimization Levels	Features	Benefits
+00 (the default)	Occurs at the machine-instruction level Constant folding Data alignment on natural boundaries Partial evaluation of test conditions Registers (simple allocation)	Compiles fastest.
+01 includes all of +00	Occurs at the block level Branch optimization Dead code elimination Instruction scheduler Peephole optimizations Registers (faster allocation)	Produces faster programs than +00, and compiles faster than level +02.

<b>Optimization Levels</b>	<b>Features</b>	<b>Benefits</b>
+O2 (-O)  <b>includes all of</b> +O0, +O1	Occurs at the routine level Common subexpression elimination Constant folding (advanced) and propagation Loop-invariant code motion Loop unrolling Registers (global allocation) Register reassociation Software pipelining Store/copy optimization Strength reduction of induction variables and constants Unused definition elimination	Can produce faster run-time code than +O1 if loops are used extensively.  Run-times for loop-oriented floating-point intensive applications may be reduced up to 90 per cent.  Operating system and interactive applications that use the optimized system libraries may achieve 30 per cent to 50 per cent additional improvement.
+O3  <b>includes all of</b> +O0 , +O1 , +O2	Occurs at the file level Cloning within a single source file Data localization Automatic and directive-specified loop parallelization Directive-specified region parallelization Directive-specified task parallelization	Can produce faster run-time code than +O2 on code that frequently calls small functions, or if loops are extensively used. Links faster than +O4.

Optimization levels  
 HP optimization levels and features

Optimization Levels	Features	Benefits
	<p>Inlining within a single source file            Loop blocking            Loop distribution            Loop fusion            Loop interchange            Loop reordering - preventing            Loop unroll and jam            Parallelization            Parallelization, preventing            Reductions            Test promotion  <i>All of the directives and pragmas of the HP parallel programming model are available in the Fortran 90 and C compilers.</i>  <code>prefer_parallel</code> requests parallelization of the following loop  <code>loop_parallel</code> forces parallelization on the last loop  <code>parallel, end_parallel</code> parallelizes a single code region to run on multiple threads.  <code>begin_tasks, next_task, end_tasks</code> forces parallelization of following code section</p>	
<p>+O4            includes all of +O0, +O1, +O2, +O3            Not available in Fortran 90</p>	<p>Occurs at the cross-module level and performed at link time            Cloning across multiple source files            Global/static variable optimizations            Inlining across multiple source files</p>	<p>Produces faster run-time code than when +O3 global variables are used or when procedure calls are inlined across modules.</p>



## Cumulative Options

The optimization options that control an optimization level are cumulative so that each option retains the optimizations of the previous option. For example, entering the following command line compiles the Fortran program `foo.f` with all `+O2`, `+O1`, and `+O0` optimizations shown in Table 2:

```
% f90 +O2 foo.f
```

In addition to these options, the `+Oparallel` option is available for use at `+O3` and above; `+Onoparallel` is the default. When the `+Oparallel` option is specified, the compiler:

- Looks for opportunities for parallel execution in loops
- Honors the parallelism-related directives and pragmas of the HP parallel programming model.

The `+Onoautopar` (no automatic parallelization) option is available for use with `+Oparallel` at `+O3` and above. `+Oautopar` is the default. `+Onoautopar` causes 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 “Parallel programming techniques,” on page 175.

## Using the Optimizer

Before exploring the various optimizations that are performed, it is important to review the coding guidelines used to assist the optimizer. This section is broken down into the following subsections:

- General guidelines
- C and C++ guidelines
- Fortran guidelines

### General guidelines

The coding guidelines presented in this section help the optimizer to optimize your program, regardless of the language in which the program is written.

- Use local variables to help the optimizer promote variables to registers.
- Do not use local variables before they are initialized. When you request +O2, +O3, or +O4 optimizations, the compiler tries to detect and indicate violations of this rule. See “+O[no]initcheck” on page 123 for related information.
- Use constants instead of variables in arithmetic expressions such as shift, multiplication, division, or remainder operations.
- Position the loop inside the procedure or use a directive to call the loop in parallel, when a loop contains a procedure call.
- Construct loops so the induction variable increases or decreases toward zero where possible. The code generated for a loop termination test is more efficient with a test against zero than with a test against some other value.
- Do not reference outside the bounds of an array. Fortran provides the -C option to check whether your program references outside array bounds.
- Do not pass an incorrect number of arguments to a function.

## C and C++ guidelines

The coding guidelines presented in this section help the optimizer to optimize your C and C++ programs.

- Use `do` loops and `for` loops in place of `while` loops. `do` loops and `for` loops are more efficient because opportunities for removing loop-invariant code are greater.
- Use `register` variables where possible.
- Use unsigned variables rather than signed, when using `short` or `char` variables or bit-fields. This is more efficient because a signed variable causes an extra instruction to be generated.
- Pass and return pointers to large structs instead of passing and returning large structs by value, where possible.
- Use type-checking tools like `lint` to help eliminate semantic errors.
- Use local variables for the upper bounds (stop values) of loops. Using local variables may enable the compiler to optimize the loop.

During optimization, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program, at least to the extent that the original unoptimized program is correct.

When gathering this information, the compiler assumes that while inside a function, the only variables that are accessed indirectly through a pointer or by another function call are:

- Global variables (all variables with file scope)
- Local variables that have had their addresses taken either explicitly by the `&` operator, or implicitly by the automatic conversion of array references to pointers.

## Optimization levels

### Using the Optimizer

In general, the preceding assumption should not pose a problem. Standard-compliant C and C++ programs do not violate this assumption. However, if you have code that does violate this assumption, the optimizer can change the behavior of the program in an undesirable way. In particular, you should follow the coding practices to ensure correct program execution for optimized code:

- Avoid using variables that are accessed by external processes. Unless a variable is declared with the `volatile` attribute, the compiler assumes that a program's data is accessed only by that program. Using the `volatile` attribute may significantly slow down a program.
- Avoid accessing an array other than the one being subscripted. For example, the construct `a[b-a]`, where `a` and `b` are the same type of array, actually references the array `b`, because it is equivalent to `*(a+(b-a))`, which is equivalent to `*b`. Using this construct might yield unexpected optimization results.
- Avoid referencing outside the bounds of the objects a pointer is pointing to. All references of the form `*(p+i)` are assumed to remain within the bounds of the variable or variables that `p` was assigned to point to.
- Do not rely on the memory layout scheme when manipulating pointers, as incorrect optimizations may result. For example, if `p` is pointing to the first member of a structure, do not assume that `p+1` points to the second member of the structure. Additionally, if `p` is pointing to the first in a list of declared variables, `p+1` is not necessarily pointing to the second variable in the list.

For more information regarding coding guidelines, see “General guidelines” on page 30.

## Fortran guidelines

The coding guidelines presented in this section help the optimizer to optimize Fortran programs.

As part of the optimization process, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program, at least to the extent that the original unoptimized program is correct.

When gathering this information, the compiler assumes that inside a routine (either a function or a subroutine) the only variables that are accessed (directly or indirectly) are:

- COMMON variables declared in the routine
- Local variables
- Parameters to this routine

Local variables include all static and nonstatic variables.

In general, you do not need to be concerned about the preceding assumption. However, if you have code that violates it, the optimizer can adversely affect the behavior of the program.

Avoid using variables that are accessed by a process other than the program. The compiler assumes that the program is the only process accessing its data. The only exception is the shared COMMON variable in Fortran 90.

Also avoid using extensive equivalencing and memory-mapping schemes, where possible.

See the section “General guidelines” on page 30 for additional guidelines.

Optimization levels  
**Using the Optimizer**

## 4 Standard optimization features

This chapter discusses the standard optimization features available with the HP-UX compilers, including those inherent in optimization levels +O0 through +O2. This includes a discussion of the following topics:

- Constant folding
- Partial evaluation of test conditions
- Simple register assignment
- Data alignment on natural boundaries
- Branch optimization
- Dead code elimination
- Faster register allocation
- Instruction scheduling
- Peephole optimizations
- Advanced constant folding and propagation
- Common subexpression elimination
- Global register allocation (GRA)
- Loop-invariant code motion, and unrolling
- Register reassociation
- Software pipelining
- Strength reduction of induction variables and constants
- Store and copy optimization
- Unused definition elimination

For more information as to specific command-line options, pragmas and directives for optimization, please see “Controlling optimization,” on page 113.

---

## Machine instruction level optimizations (+O0)

At optimization level +O0, the compiler performs optimizations that span only a single source statement. This is the default. The +O0 machine instruction level optimizations include:

- Constant folding
- Partial evaluation of test conditions
- Simple register assignment
- Data alignment on natural boundaries

### Constant folding

Constant folding is the replacement of operations on constants with the result of the operation. For example,  $Y=5+7$  is replaced with  $Y=12$ .

More advanced constant folding is performed at optimization level +O2. See the section “Advanced constant folding and propagation” on page 42 for more information.

### Partial evaluation of test conditions

Where possible, the compiler determines the truth value of a logical expression without evaluating all the operands. This is known as short-circuiting. The Fortran example below describes this:

```
IF ((I .EQ. J) .OR. (I .EQ. K)) GOTO 100
```

If  $(I .EQ. J)$  is true, control immediately goes to 100; otherwise,  $(I .EQ. K)$  must be evaluated before control can go to 100 or the following statement.

Do not rely upon partial evaluation if you use function calls in the logical expression because:

- There is no guarantee on the order of evaluation.
- A procedure or function call can have side effects on variable values that may or may not be partially evaluated correctly.



## Simple register assignment

The compiler may place frequently used variables in registers to avoid more costly accesses to memory.

A more advanced register assignment algorithm is used at optimization level +O2. See the section “Global register allocation (GRA)” on page 43 for more information.

## Data alignment on natural boundaries

The compiler automatically aligns data objects to their natural boundaries in memory, providing more efficient access to data. This means that a data object’s address is integrally divisible by the length of its data type; for example, `REAL*8` objects have addresses integrally divisible by 8 bytes.

### NOTE

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

Declare scalar variables in order from longest to shortest data length to ensure the efficient layout of such aligned data in memory. This minimizes the amount of padding the compiler has to do to get the data onto its natural boundary.

### Example

#### Data alignment on natural boundaries

The following Fortran example describes the alignment of data objects to their natural boundaries:

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

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

Standard optimization features

Machine instruction level optimizations (+O0)

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 performed on all data. This is not to be confused with cache line boundary alignment.

## Block level optimizations (+O1)

At optimization level +O1, the compiler performs optimizations on a block level. The compiler continues to run the +O0 optimizations, with the following additions:

- Branch optimization
- Dead code elimination
- Faster register allocation
- Instruction scheduling
- Peephole optimizations

### Branch optimization

Branch optimization involves traversing the procedure and transforming branch instruction sequences into more efficient sequences where possible. Examples of possible transformations are:

- Deleting branches whose target is the fall-through instruction (the target is two instructions away)
- Changing the target of the first branch to be the target of the second (unconditional) branch when the target of a branch is an unconditional branch
- Transforming an unconditional branch at the bottom of a loop that branches to a conditional branch at the top of the loop into a conditional branch at the bottom of the loop
- Changing an unconditional branch to the exit of a procedure into an exit sequence where possible
- Changing conditional or unconditional branch instructions that branch over a single instruction into a conditional nullification in the following instruction
- Looking for conditional branches over unconditional branches, where the sense of the first branch could be inverted and the second branch deleted. These result from null THEN clauses and from THEN clauses that only contain GOTO statements.

## Example

### Conditional/unconditional branches

The following Fortran example provides a transformation from a branch instruction to a more efficient sequence:

```
      IF (L) THEN  
        A=A*2  
      ELSE  
        GOTO 100  
      ENDIF  
      B=A+1  
100   C=A*10
```

becomes:

```
      IF (.NOT. L) GOTO 100  
      A=A*2  
      B=A+1  
100   C=A*10
```

### Dead code elimination

Dead code elimination removes unreachable code that is never executed.

For example, in C:

```
if(0)  
  a = 1;  
else  
  a = 2;
```

becomes:

```
  a = 2;
```

### Faster register allocation

Faster register allocation involves:

- Inserting entry and exit code
- Generating code for operations such as multiplication and division
- Eliminating unnecessary copy instructions
- Allocating actual registers to the dummy registers in instructions

Faster register allocation, when used at +O0 or +O1, analyzes register use faster than the global register allocation performed at +O2.

## Instruction scheduling

The instruction scheduler optimization performs the following tasks:

- Reorders the instructions in a basic block to improve memory pipelining. For example, where possible, a load instruction is separated from the use of the loaded register.
- Follows a branch instruction with an instruction that is executed as the branch occurs, where possible.
- Schedules floating-point instructions.

## Peephole optimizations

A peephole optimization is a machine-dependent optimization that makes a pass through low-level assembly-like instruction sequences of the program. It applies patterns to a small window (peephole) of code looking for optimization opportunities. It performs the following optimizations:

- Changes the addressing mode of instructions so they use shorter sequences
- Replaces low-level assembly-like instruction sequences with faster (usually shorter) sequences and removes redundant register loads and stores

## Routine level optimizations (+O2)

At optimization level +O2, the compiler performs optimizations on a routine level. The compiler continues to perform the optimizations performed at +O1, with the following additions:

- Advanced constant folding and propagation
- Common subexpression elimination
- Global register allocation (GRA)
- Loop-invariant code motion
- Loop unrolling
- Register reassociation
- Software pipelining
- Strength reduction of induction variables and constants
- Store and copy optimization
- Unused definition elimination

### Advanced constant folding and propagation

Constant folding computes the value of a constant expression at compile time. Constant propagation is the automatic compile-time replacement of variable references with a constant value previously assigned to that variable.

#### Example

#### Advanced constant folding and propagation

The following C/C++ code example describes an advanced constant folding and propagation:

```
a = 10;  
b = a + 5;  
c = 4 * b;
```

Once `a` is assigned, its value is propagated to the statement where `b` is assigned so that the assignment reads:

```
b = 10 + 5;
```

The expression  $10 + 5$  can then be folded. Now that  $b$  has been assigned a constant, the value of  $b$  is propagated to the statement where  $c$  is assigned. After all the folding and propagation, the original code is replaced by:

```
a = 10;  
b = 15;  
c = 60;
```

## Common subexpression elimination

Common subexpression elimination optimization identifies expressions that appear more than once and have the same result. It then computes the result and substitutes the result for each occurrence of the expression. Subexpression types include instructions that load values from memory, as well as arithmetic evaluation.

### Example

#### Common subexpression elimination

In Fortran, for example, the code first looks like this:

```
A = X + Y + Z  
B = X + Y + W
```

After this form of optimization, it becomes:

```
T1 = X + Y  
A = T1 + Z  
B = T1 + W
```

## Global register allocation (GRA)

Scalar variables can often be stored in registers, eliminating the need for costly 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.

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

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

Parallel assignments to the same shared variables from multiple threads make sense only if the assignments are contained inside critical or ordered sections, or are executed conditionally based on the thread ID. GRA does 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 12, “Parallel synchronization” for a discussion of `sync_routine`). However, for conditional assignments based on the thread ID, GRA may allocate registers that may cause wrong answers when stored.

In such cases, GRA is disabled only for shared variables that are visible to multiple threads by specifying `+Onosharedgra`. A description of this option is located in “+O[no]sharedgra” on page 138.

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 predetermined limit. You can remove this limit (and possibly increase compile time) by specifying `+O[no]limit`. A description of this option is located in “+O[no]limit” on page 126.

This optimization is also known as coloring register allocation because of the similarity to map-coloring algorithms in graph theory.

## Register allocation in C and C++

In C and C++, you can help the optimizer understand when certain variables are heavily used within a function by declaring these variables with the `register` qualifier.

GRA may override your choices and promote a variable not declared `register` to a register over a variable that is declared `register`, based on estimated speed improvements.



## Loop-invariant code motion

The loop-invariant code motion optimization recognizes instructions inside a loop whose results do not change and then moves the instructions outside the loop. This optimization ensures that the invariant code is only executed once.

### Example

#### Loop-invariant code motion

This example begins with following C/C++ code:

```
x = z;  
for(i=0; i<10; i++)  
    a[i] = 4 * x + i;
```

After loop-invariant code motion, it becomes:

```
x = z;  
t1 = 4 * x;  
for(i=0; i<10; i++)  
    a[i] = t1 + i;
```

## Loop unrolling

Loop unrolling increases a loop's step value and replicates the loop body. Each replication is appropriately offset from the induction variable so that all iterations are performed, given the new step.

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

Loop unrolling and the unroll factor are controlled using the `+O[no]loop_unroll[=unroll factor]`. This option is described on page 127.

Some loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, loop replication is limited by default. You can increase this limit (and possibly increase your program's compile time and code size) by specifying the `+Onosize` and `+Onolimit` compiler options.

### Example

#### Loop unrolling

Consider the following Fortran example:

```
SUBROUTINE FOO(A,B)  
REAL A(10,10), B(10,10)  
DO J=1, 4
```

Standard optimization features  
Routine level optimizations (+O2)

```
DO I=1, 4
  A(I,J) = B(I,J)
ENDDO
ENDDO
END
```

The loop nest is completely unrolled as shown below:

```
A(1,1) = B(1,1)
A(2,1) = B(2,1)
A(3,1) = B(3,1)
A(4,1) = B(4,1)
```

```
A(1,2) = B(1,2)
A(2,2) = B(2,2)
A(3,2) = B(3,2)
A(4,2) = B(4,2)
```

```
A(1,3) = B(1,3)
A(2,3) = B(2,3)
A(3,3) = B(3,3)
A(4,3) = B(4,3)
```

```
A(1,4) = B(1,4)
A(2,4) = B(2,4)
A(3,4) = B(3,4)
A(4,4) = B(4,4)
```

Partial unrolling is performed on loops with larger or unknown iteration counts. This form of unrolling retains the loop structure, but replicates the body a number of times equal to the unroll factor and adjusts references to the iteration variable accordingly.

**Example**

**Loop unrolling**

This example begins with the following Fortran example:

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

It is unrolled to a depth of four 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 four values of A instead of one value. The compiler also generates 'clean-up' code for the case where the range is not evenly divisible by the unroll factor.

## Register reassociation

Array references often require one or more instructions to compute the virtual memory address of the array element specified by the subscript expression. The register reassociation optimization implemented in PA-RISC compilers tries to reduce the cost of computing the virtual memory address expression for array references found in loops.

Within loops, the virtual memory address expression is rearranged and separated into a loop-variant term and a loop-invariant term.

- Loop-variant terms are those items whose values may change from one iteration of the loop to another.
- Loop-invariant terms are those items whose values are constant throughout all iterations of the loop. The loop-variant term corresponds to the difference in the virtual memory address associated with a particular array reference from one iteration of the loop to the next.

The register reassociation optimization dedicates a register to track the value of the virtual memory address expression for one or more array references in a loop and updates the register appropriately in each iteration of a loop.

The register is initialized outside the loop to the loop-invariant portion of the virtual memory address expression. The register is incremented or decremented within the loop by the loop-variant portion of the virtual memory address expression. The net result is that array references in loops are converted into equivalent, but more efficient, pointer dereferences.

Register reassociation can often enable another loop optimization. After performing the register reassociation optimization, the loop variable may be needed only to control the iteration count of the loop. If this is the case, the original loop variable is eliminated altogether by using the PA-RISC `ADDIB` and `ADDB` machine instructions to control the loop iteration count.

You can enable or disable register reassociation using the `+O[no]regreassoc` command-line option at +O2 and above. The default is `+Oregreassoc`. See “`+O[no]regreassoc`” on page 136 for more information.

## Example

### Register allocation

This example begins with the following C/C++ code:

```
int a[10][20][30];

void example (void)
{
    int i, j, k;

    for (k = 0; k < 10; k++)
        for (j = 0; j < 10; j++)
            for (i = 0; i < 10; i++)
                a[i][j][k] = 1;
}
```

After register reassociation is applied, the innermost loop becomes:

```
int a[10][20][30];

void example (void)
{
    int i, j, k;
    register int (*p)[20][30];

    for (k = 0; k < 10; k++)
        for (j = 0; j < 10; j++)
            for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10;
i++)
                *(p++[0][0]) = 1;
}
```

As you can see, the compiler-generated temporary register variable, `p`, strides through the array `a` in the innermost loop. This register pointer variable is initialized outside the innermost loop and auto-incremented within the innermost loop as a side-effect of the pointer dereference.

## Software pipelining

Software pipelining transforms code in order to optimize program loops. It achieves this by rearranging the order in which instructions are executed in a loop. Software pipelining generates code that overlaps operations from different loop iterations. It is particularly useful for loops that contain arithmetic operations on `REAL*4` and `REAL*8` data in Fortran or on `float` and `double` data in C or C++.

The goal of this optimization is to avoid processor stalls due to memory or hardware pipeline latencies. The software pipelining transformation partially unrolls a loop and adds code before and after the loop to achieve a high degree of optimization within the loop.

You can enable or disable software pipelining using the `+O[no]pipeline` command-line option at +O2 and above. The default is `+Opipeline`. Use `+Onopipeline` if a smaller program size and faster compile time are more important than faster execution speed. See “`+O[no]pipeline`” on page 130 for more information.

### Prerequisites of pipelining

Software pipelining is attempted on a loop that meets the following criteria:

- It is the innermost loop
- There are no branches or function calls within the loop
- The loop is of moderate size

This optimization produces slightly larger program files and increases compile time. It is most beneficial in programs containing loops that are executed many times.

### Example

#### Software pipelining

The following C/C++ example shows a loop before and after the software pipelining optimization:

```
#define SIZ 10000
float x[SIZ], y[SIZ];
int i;
init();
for (i = 0; i <= SIZ; i++)
    x[i] = x[i] / y[i] + 4.00;
```

Standard optimization features  
Routine level optimizations (+O2)

Four significant things happen in this example:

- A portion of the first iteration of the loop is performed before the loop.
- A portion of the last iteration of the loop is performed after the loop.
- The loop is unrolled twice.
- Operations from different loop iterations are interleaved with each other.

When this loop is compiled with software pipelining, the optimization is expressed as follows:

R1 = 0;	Initialize array index
R2 = 4.00;	Load constant value
R3 = X[0];	Load first X value
R4 = Y[0];	Load first Y value
R5 = R3 / R4;	Perform division on first element: $n = X[0]/Y[0]$
do {	Begin loop
R6 = R1;	Save current array index
R1++;	Increment array index
R7 = X[R1];	Load current X value
R8 = Y[R1];	Load current Y value
R9 = R5 + R2;	Perform addition on prior row: $X[i] = n + 4.00$
R10 = R7 / R8;	Perform division on current row: $m = X[i+1]/Y[i+1]$
X[R6] = R9;	Save result of operations on prior row
R6 = R1;	Save current array index
R1++;	Increment array index
R3 = X[R1];	Load next X value
R4 = Y[R1];	Load next Y value

```
R11 = R10 + R2;   Perform addition on current row:
                  X[i+1] = m + 4.00
R5 = R3 / R4;     Perform division on next row: n =
                  X[i+2]/Y[i+2]
X[R6] = R11;      Save result of operations on current row
} while (R1 <= 100); End loop
R9 = R5 + R2;     Perform addition on last row: X[i+2] =
                  n + 4.00
X[R6] = R9;       Save result of operations on last row
```

This transformation stores intermediate results of the division instructions in unique registers (noted as *n* and *m*). These registers are not referenced until several instructions after the division operations. This decreases the possibility that the long latency period of the division instructions will stall the instruction pipeline and cause processing delays.

## Strength reduction of induction variables and constants

This optimization removes expressions that are linear functions of a loop counter and replaces each of them with a variable that contains the value of the function. Variables of the same linear function are computed only once. This optimization also replaces multiplication instructions with addition instructions wherever possible.

### Example

#### Strength reduction of induction variables and constants

This example begins with the following C/C++ code:

```
for (i=0; i<25; i++) {
    r[i] = i * k;
}
```

After this optimization, it looks like this:

```
t1 = 0;
for (i=0; i<25; i++) {
    r[i] = t1;
    t1 += k;
}
```

## Store and copy optimization

Where possible, the store and copy optimization substitutes registers for memory locations, by replacing store instructions with copy instructions and deleting load instructions.

## Unused definition elimination

The unused definition elimination optimization removes unused memory location and register definitions. These definitions are often a result of transformations made by other optimizations.

### Example

#### Unused definition elimination

This example begins with the following C/C++ code:

```
f(int x){
    int a,b,c;

    a = 1;
    b = 2;
    c = x * b;
    return c;
}
```

After unused definition elimination, it looks like this:

```
f(int x) {
    int a,b,c;

    c = x * 2;
    return c;
}
```

The assignment `a = 1` is removed because `a` is not used after it is defined. Due to another +O2 optimization (constant propagation), the `c = x * b` statement becomes `c = x * 2`. The assignment `b = 2` is then removed as well.



# 5 **Loop and cross-module optimization features**

This chapter discusses loop optimization features available with the HP-UX compilers, including those inherent in optimization level +O3. This includes a discussion of the following topics:

- Strip mining
- Inlining within a single source file
- Cloning within a single source file
- Data localization
- Loop blocking
- Loop distribution
- Loop fusion
- Loop interchange
- Loop unroll and jam
- Preventing loop reordering
- Test promotion
- Cross-module cloning

For more information as to specific loop optimization command-line options, as well as related pragmas and directives for optimization, please see “Controlling optimization,” on page 113.

---

## Strip mining

Strip mining is a fundamental  $\text{O}3$  transformation. Used by itself, strip mining is not profitable. However, it is used by loop blocking, loop unroll and jam, 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, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's strip length.

### Example

#### Strip mining

This example begins with the Fortran code below:

```
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 IOITER = 1, 10000, 1000
  DO ISTRIP = IOITER, IOITER+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—if  $I$  went from 1 to 10500 rather than 1 to 10000, for example—the final iteration of the strip loop would execute 500 iterations instead of 1000.

---

## Inlining within a single source file

Inlining substitutes selected function calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. Inlining may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns.

At +O3, inlining is performed within a file; at +O4, it is performed across files. Inlining is affected by the +O[no]inline[=*namelist*] and +Oinline\_budget=*n* command-line options. See "Controlling optimization," on page 113 for more information.

### Example

#### Inlining within single source file

The following is an example of inlining at the source code level. Before inlining, the C source file looks like this:

```
/* Return the greatest common divisor of two positive integers,*/
/* int1 and int2, computed using Euclid's algorithm. (Return 0 */
/* if either is not positive.) */

int gcd(int int1,int int2)
{
    int inttemp;

    if ( (int1 <= 0) || (int2 <= 0) ) {
        return(0);
    }
    do {
        if (int1 < int2) {
            inttemp = int1;
            int1     = int2;
            int2     = inttemp;
        }
        int1 = int1 - int2;
    } while (int1 > 0);
    return(int2);
}

main()
{
    int xval,yval,gcdxy;
    .
    .      /* statements before call to gcd */
    .
    gcdxy = gcd(xval,yval);
    .
    .      /* statements after call to gcd */
    .
}
```

Loop and cross-module optimization features  
Inlining within a single source file

After inlining, main looks like this:

```
main()
{
  int xval,yval,gcdxy;
  .          /* statements before inlined version of gcd */
  ;
  {
    int int1;
    int int2;

    int1 = xval;
    int2 = yval;
    {
      int inttemp;

      if ( (int1 <= 0) || (int2 <= 0) ){
        gcdxy = (0);
        goto AA003;
      }
      do {
        if (int1 < int2){
          inttemp = int1;
          int1     = int2;
          int2     = inttemp;
        }
        int1 = int1 - int2;
      } while (int1 > 0);
      gcdxy = (int2);
    }
  }
AA003 : ;
  .          /* statements after inlined version of gcd */
  ;
}
```

## Cloning within a single source file

Cloning replaces a call to a routine by calling a clone of that routine. The clone is optimized differently than the original routine.

Cloning can expose additional opportunities for interprocedural optimization. At `+O3`, cloning is performed within a file, and at `+O4`, cloning is performed across files. Cloning is enabled by default, and is disabled by specifying the `+Onoinline` command-line option.

---

## Data localization

Data localization occurs as a result of various loop transformations that occur at optimization levels +02 or +03. Because optimizations are cumulative, specifying +03 or +04 takes advantage of the transformations that happen at +02.

**Table 3** Loop transformations affecting data localization

Loop transformation	Options required for behavior to occur
Loop unrolling	+02 +Oloop_unroll (+Oloop_unroll is on by default at +02 and above)
Loop distribution	+03 +Oloop_transform (+Oloop_transform is on by default at +03 and above)
Loop interchange	+03 +Oloop_transform (+Oloop_transform is on by default at +03 and above)
Loop blocking	+03 +Oloop_transform +Oloop_block (+Oloop_transform is on by default at +03 and above) (+Oloop_block is off by default)
Loop fusion	+03 +Oloop_transform (+Oloop_transform is on by default at +03 and above)
Loop unroll and jam	+03 +Oloop_transform +Oloop_unroll_jam (+Oloop_transform is on by default at +03 and above) (+Oloop_unroll_jam is off by default at +03 and above)

Data localization keeps frequently used data in the processor data cache, eliminating the need for more costly memory accesses.

Loops that manipulate arrays are the main candidates for localization optimizations. Most of these loops are eligible for the various transformations that the compiler performs at +03. These transformations are explained in detail in this section.

Some loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, loop replication is limited by default. You can increase this limit (and possibly increase your program's compile time and code size) by specifying the `+Onosize` and `+Onolimit` compiler options.

**NOTE**

Most of the following code examples demonstrate optimization by showing the original code first and optimized code second. The optimized code is shown in the same language as the original code for illustrative purposes only.

## Conditions that inhibit data localization

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

- Loop-carried dependences (LCDs)
- Other loop fusion dependences
- Aliasing
- Computed or assigned GOTO statements in Fortran
- `return` or `exit` statements in C or C++
- `throw` statements in C++
- Procedure calls

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

### Loop-carried dependences (LCDs)

A loop-carried dependence (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 that are offset in opposite directions.

To ignore LCDs, use the `no_loop_dependence` directive or `pragma`. The form of this directive and `pragma` is shown in Table 4.

**NOTE**

This directive and `pragmas` should only be used if you are certain that there are no loop dependences. Otherwise, errors will result.

**Table 4**

**Form of `no_loop_dependence` directive and `pragma`**

Language	Form
Fortran	<code>C\$DIR NO_LOOP_DEPENDENCE ( <i>namelist</i> )</code>
C	<code>#pragma _CNX no_loop_dependence ( <i>namelist</i> )</code>

where

*namelist* is a comma-separated list of variables or arrays that have no dependences for the immediately following loop.

**Example**

**Loop-carried dependences**

The Fortran loop below contains an LCD that inhibits interchange:

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

C and C++ loops can contain similar constructs, but to simplify illustration, only the Fortran example is discussed here.

As written, this loop uses  $A(I-1, J-1)$  and  $A(I-1, J+1)$  to compute  $A(I, J)$ . Table 5 shows the sequence in which values of  $A$  are computed for this loop.



**Table 5** Computation sequence of  $A(I, J)$ : original loop

$I$	$J$	$A(I, J)$	$A(I-1, J-1)$	$A(I-1, J+1)$
2	2	$A(2, 2)$	$A(1, 1)$	$A(1, 3)$
2	3	$A(2, 3)$	$A(1, 2)$	$A(1, 4)$
2	4	$A(2, 4)$	$A(1, 3)$	$A(1, 5)$
...	...	...	...	...
3	2	$A(3, 2)$	$A(2, 1)$	$A(2, 3)$
3	3	$A(3, 3)$	$A(2, 2)$	$A(2, 4)$
3	4	$A(3, 4)$	$A(2, 3)$	$A(2, 5)$
...	...	...	...	...

As shown in Table 5, 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 dependence must be honored for the loop to yield its intended results. If a row element of  $A$  is computed before the previous row elements are computed, the result is incorrect.

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

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

After interchange, the loop computes values of  $A$  in the sequence shown in Table 6.

**Table 6** Computation sequence of  $A(I, J)$ : interchanged loop

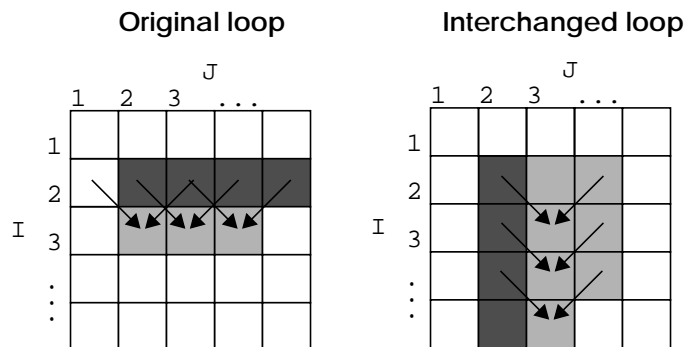
I	J	$A(I, J)$	$A(I-1, J-1)$	$A(I-1, J+1)$
2	2	$A(2, 2)$	$A(1, 1)$	$A(1, 3)$
3	2	$A(3, 2)$	$A(2, 1)$	$A(2, 3)$
4	2	$A(4, 2)$	$A(3, 1)$	$A(3, 3)$
...	...	...	...	...
2	3	$A(2, 3)$	$A(1, 2)$	$A(1, 4)$
3	3	$A(3, 3)$	$A(2, 2)$	$A(2, 4)$
4	3	$A(4, 3)$	$A(3, 2)$	$A(3, 4)$
...	...	...	...	...

Here, the elements of the current column of  $A$  are computed using the elements of the previous column and the next column of  $A$ .

The problem here is that columns of  $A$  are being computed using elements from the next column, which have not been written yet. This computation violates the dependence illustrated in Table 5.

The element-to-element dependences in both the original and interchanged loop are illustrated in Figure 9.

**Figure 9** LCDs in original and interchanged loops



The arrows in Figure 9 represent dependences from one element to another, pointing at elements that depend on the elements' bases. Shaded elements indicate a typical row or column computed in the inner loop:

- Darkly shaded elements have already been computed.
- Lightly shaded elements have not yet 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.

## Example

### Avoid loop interchange

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

Occasionally, the compiler encounters an apparent LCD. If it cannot determine whether the LCD actually inhibits interchange, it conservatively avoids 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
```

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

If you are positive that the IADD and JADD are both negative or both positive, you can tell the compiler that the loop is free of dependences using the `no_loop_dependence` directive or pragma, described in this chapter Table 4 on page 60.

The previous Fortran loop is interchanged when the `NO_LOOP_DEPENDENCE` directive is specified for A 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
```

### Data localization

If `IADD` and `JADD` acquire opposite-signed values at runtime, these loops may result in incorrect answers.

## Other loop fusion dependences

In some cases, loop fusion is also inhibited by simpler dependences 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
```

While it might appear that loop fusion would benefit the preceding example, it would actually 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 has not 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.

## Aliasing

An alias is an alternate name for an 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. The use of pointers normally causes the problem in C and C++. Passing identical actual arguments into different dummy arguments in a Fortran subprogram can also cause aliasing, as can passing the same address into different pointer arguments in a C or C++ function.

### Example

#### Aliasing

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

```
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; because 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 and C++ equivalent of this loop follows. Keep in mind that C and C++ store 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];
  }
}
```

Fortran's EQUIVALENCE statement is imitated in C and C++; through the use of pointers, arrays are effectively equivalenced, as shown.

Passing the same address into different dummy procedure arguments can yield the same result. Fortran passes arguments by reference while C and C++ pass them by value. However, pass-by-reference is simulated in C and C++ by passing the argument's address into a pointer in the receiving procedure or in C++ by using references.

### Example

#### Aliasing

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

### NOTE

The sample code below violates the Fortran standard.

```
.
.
.
CALL ALI(A,A,C)
.
.
.
SUBROUTINE ALI(A,B,C)
INTEGER A(100,100), B(100,100), C(100,100)
```

## Loop and cross-module optimization features

### Data localization

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

The following (legal ANSI C) code shows the same argument-passing problem in 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];
    }
  }
}
```

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

### I/O statements

The order in which values are read into or written from a loop may change if the loop is interchanged. For this reason, I/O statements inhibit interchange and, consequently, data localization.

#### Example

#### I/O statements

The following Fortran code is the basis for this example:

```
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 10.

Figure 10

#### Values read into array A

		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

## 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, however. These are often created in C and C++ using the `break` statement, and in Fortran using the `GOTO` statement.

As noted before, the order of computation changes if the loops are interchanged.

### Example

#### Multiple loop entries or exits

This example begins with 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 computes. Interchange therefore may cause the results of the loop to differ and must be avoided.

## 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.

## return or exit statements in C or C++

Similar to Fortran's `RETURN` and `STOP` statements (discussed in the previous section), `return` and `exit` statements in C and C++ inhibit localization because they inhibit interchange.



### **throw statements in C++**

In C++, `throw` statements, like loops containing multiple exits, inhibit localization because they inhibit interchange.

### **Procedure calls**

HP compilers are unaware of the side effects of most procedures, and therefore cannot determine whether or not they might interfere with loop interchange. Consequently, the compilers do not perform loop interchange in an embedded procedure call. These side effects may include data dependences involving loop arrays, aliasing (as described in the section “Aliasing” on page 64), and processor data cache that use conflicts with the loop’s cache. This renders useless any data localization optimizations performed on the loop.

#### **NOTE**

The compiler can loop parallel on a loop with a procedure call if it can verify that the procedure will not cause any side effects.

---

## 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 optimally sized to fit in the cache.

The loop-blocking optimization is only available at +O3 (and above) in the HP compilers; it is disabled by default. To enable loop blocking, use the +Oloop\_block option. Specifying +Onoloop\_block (the default) disables both automatic and directive-specified loop blocking. Specifying +Onoloop\_transform also disables loop blocking, as well as loop distribution, loop interchange, loop fusion, loop unroll, and loop unroll and jam.

Loop blocking can also be enabled for specific loops using the block\_loop directive and pragma. The block\_loop and no\_block\_loop directives and pragmas affect the immediately following loop. You can also instruct the compiler to use a specific block factor using block\_loop. The no\_block\_loop directive and pragma disables loop blocking for a particular loop.

The forms of these directives and pragmas is shown in Table 7.

**Table 7**                    **Forms of block\_loop, no\_block\_loop directives and pragmas**

Language	Form
Fortran	C\$DIR BLOCK_LOOP[(BLOCK_FACTOR = <i>n</i> )]  C\$DIR NO_BLOCK_LOOP
C	#pragma _CNX block_loop[(block_factor = <i>n</i> )]  #pragma _CNX no_block_loop

where

$n$  is the requested block factor, which must be a compile-time integer constant. The compiler uses this value as stated. For the 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 the absence of the `block_factor` argument, this directive is useful for indicating which loop in a nest to block. In this case, the compiler uses a heuristic to determine the block factor.

## Data reuse

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

- Spatial reuse
- Temporal reuse

### Spatial reuse

Spatial reuse uses data that was encached as a result of fetching another piece of data from memory; data is fetched by cache lines. 32 bytes of data is encached on every fetch on V2250 servers. Cache line sizes may be different on other HP SMPs.

On the initial fetch of array data from memory within a stride-one loop, the requested item is located anywhere in the 32 bytes. The exception is if array is aligned on cache line boundaries. Refer to “Standard optimization features,” on page 35, for a description of data alignment.

Starting with the cache-aligned memory fetch, the requested data is located at the beginning of the cache line, and the rest of the cache line contains subsequent array elements. For a `REAL*4` array, this means the requested element and the seven following elements are encached on each fetch after the first.

If any of these seven elements could then be used 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 contain fewer usable elements.

## Temporal reuse

Temporal reuse uses 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 that stride through arrays are candidates for blocking when there is also an outermost loop carrying spatial or temporal reuse. Blocking the innermost loop allows data referenced by the outer loop to remain in the cache across multiple iterations. 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.

### Example

#### Simple loop blocking

In order to exploit reuse in more realistic examples that manipulate arrays that do not all fit in the cache, the compiler can apply a blocking transformation.

The following Fortran example demonstrates this:

```
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. Thus, blocking becomes profitable.

First the compiler strip mines the `I` loop:

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

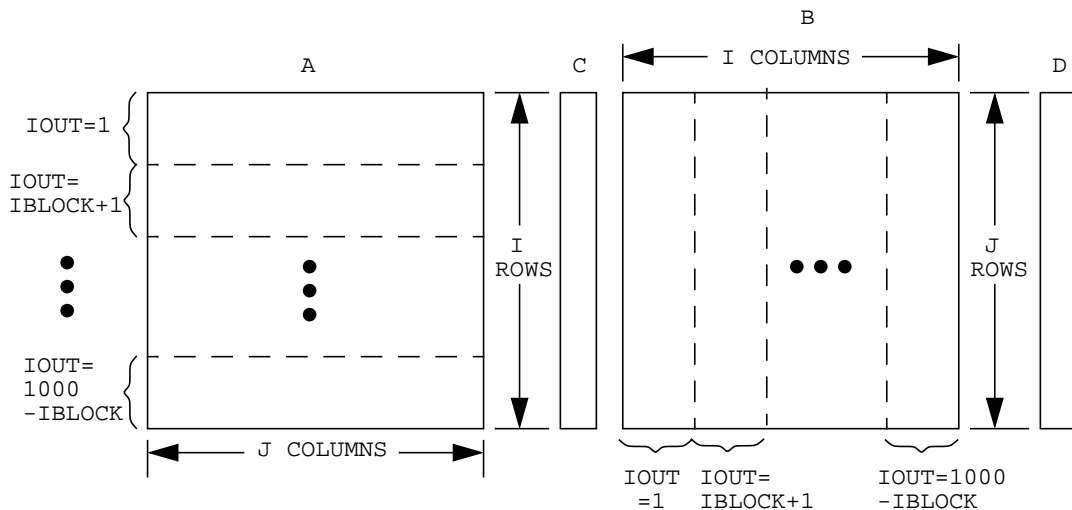
`IBLOCK` is the block factor (also referred to as the strip mine length) the compiler chooses based on the size of the arrays and size of the cache. Note that this example assumes the chosen `IBLOCK` divides 1000 evenly.

Next, the compiler moves the outer strip loop (IOUT) outward as far as possible.

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

This new nest accesses IBLOCK rows of A and IBLOCK columns of B for every iteration of J. At every iteration of IOUT, 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 11.

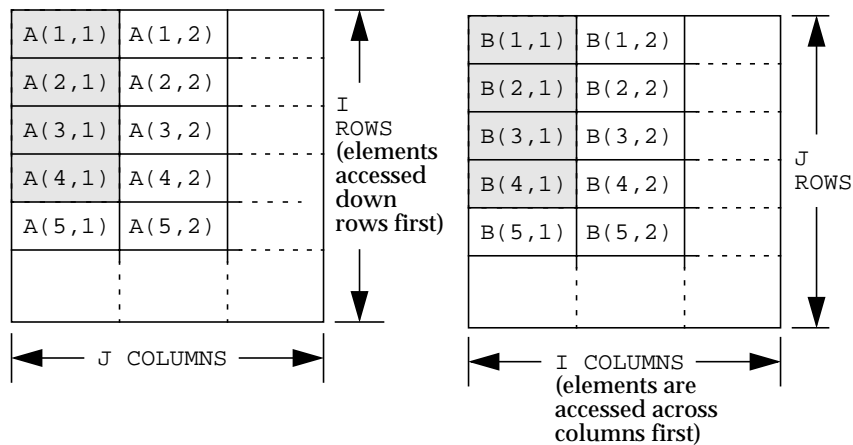
**Figure 11**      **Blocked array access**



Fetches of A encache the needed element and the three elements that are used in the three subsequent iterations, giving spatial reuse on A. Because the I loop traverses columns of B, fetches of B encache extra elements that are not spatially reused until J increments. IBLOCK is chosen by the compiler to efficiently exploit spatial reuse of both A and B.

Figure 12 illustrates how cache lines of each array are fetched. A and B both start on cache line boundaries because they are in COMMON. The shaded area represents the initial cache line fetched.

**Figure 12**      **Spatial reuse of A and B**



- When A(1,1) is accessed, A(1:4,1) is fetched; A(2:4,1) is used on subsequent iterations 2,3 and 4 of I.
- B(1:4,1) is fetched when I=1, but B(2:4,1) is not be used until J increments to 2, 3, 4. B(1:4,2) is fetched when I=2.

Typically, IBLOCK elements of C 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 removed from the I loop so that it remains in a register for all iterations of I.

**Example**

**Matrix multiply blocking**

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
```

As a result, the following occurs:

- 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
- Temporal reuse of C with respect to the K loop

An analogous C and C++ example follows with a different resulting interchange:

```
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];
```

The HP C and aC++ compilers interchange and block the loop in this example to provide optimal access efficiency for the row-major C and 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];
```

### Loop blocking

As you can see, the interchange was done differently because of C and C++'s different array storage strategies. This code yields:

- Spatial reuse of `b` with respect to the `j` loop
- Temporal reuse of `b` with respect to the `i` 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
- 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 is not performed.

### Example

#### Loop blocking

The following example shows the affect of the `block_loop` directive on the code shown earlier in “Matrix multiply blocking” on page 74:

```

REAL*8 A(1000,1000),B(1000,1000)
REAL*8 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

```

The original example involving this code showed 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 are not necessary. The compiler-chosen value is still used on the `I` loop.



## Loop distribution

Loop distribution is another fundamental +O3 transformation necessary for 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 that contain all computations inside the body of the innermost loop.

Loop distribution takes place at +O3 and above and is enabled by default. Specifying +Onolooop\_transform disables loop distribution, as well as loop interchange, loop blocking, loop fusion, loop unroll, and loop unroll and jam.

Loop distribution is disabled for specific loops by specifying the no\_distribute directive or pragma immediately before the loop.

The form of this directive and pragma is shown in Table 8.

**Table 8**

**Form of no\_distribute directive and pragma**

Language	Form
Fortran	C\$DIR NO_DISTRIBUTE
C	#pragma _CNX no_distribute

**Example**

### Loop distribution

This example begins with the following Fortran code:

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

Loop distribution creates two copies of the I loop, separating the nested J loop from the assignments to array C. In this way, all assignments are moved to innermost loops. Interchange is then performed on the I and J loops.

## Loop and cross-module optimization features

### Loop distribution

The distribution and interchange is shown in the following transformed code:

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

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

---

## 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 increases register usage. It can also lead to other optimizations. By potentially reducing the number of parallelizable loops in a program and increasing the amount of work in each of those loops, loop fusion can greatly reduce parallelization overhead. Because fewer spawns and joins are necessary.

Loop fusion takes place at +O3 and above and is enabled by default. Specifying +OnolooP\_transform disables loop fusion, as well as loop distribution, loop interchange, loop blocking, loop unroll, and loop unroll and jam.

Occasionally, loops that do not appear to be fusible become fusible as a result of compiler transformations that precede fusion. For instance, 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 inhibit fusion, they are fused.

### Example

#### Loop fusion

This example begins with 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
```

The two loops shown above are fused into the following loop using loop fusion:

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

## Example

### Loop fusion

This example begins with the following Fortran code:

```
REAL A(100,100), B(100,100), C(100,100)
.
.
.
C = 2.0 * B
A = A + B
```

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

```
DO TEMP1 = 1, 100
  DO TEMP2 = 1, 100
    C(TEMP2, TEMP1) = 2.0 * B(TEMP2, TEMP1)
  ENDDO
ENDDO
DO TEMP3 = 1, 100
  DO TEMP4 = 1, 100
    A(TEMP4, TEMP3) = A(TEMP4, TEMP3) + B(TEMP4, TEMP3)
  ENDDO
ENDDO
```

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

```
DO TEMP1 = 1, 100
  DO TEMP2 = 1, 100
    C(TEMP2, TEMP1) = 2.0 * B(TEMP2, TEMP1)
    A(TEMP2, TEMP1) = A(TEMP2, TEMP1) + B(TEMP2, TEMP1)
  ENDDO
ENDDO
```

Further optimizations could be applied to this new nest as appropriate.

## Example

### Loop peeling

When trip counts of adjacent loops differ by only a single iteration (+1 or -1), the compiler may peel an iteration from one of the two loops so that the loops may then be fused. The peeled iteration is performed separately from the original loop.

The following Fortran example shows how this is implemented:

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

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

As you can see, the  $N$ th iteration of the  $J$  loop is peeled, resulting in a trip count of  $N - 1$ . The  $N$ th iteration is performed outside the  $J$  loop. As a result, the code is changed to the following:

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

DO J = 1, N-1
  A(J) = A(J) + 1
ENDDO

A(N) = A(N) + 1
```

The  $I$  and  $J$  loops now have the same trip count and are fused, as shown below:

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

A(N) = A(N) + 1
```

## Loop interchange

The compiler may interchange (or reorder) 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
- To optimize inner-loop memory accesses

Loop interchange takes place at +O3 and above and is enabled by default. Specifying `+Onolloop_transform` disables loop interchange, as well as loop distribution, loop blocking, loop fusion, loop unroll, and loop unroll and jam.

### Example

#### Loop interchange

This example begins with the Fortran matrix addition algorithm below:

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

The 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, facilitates 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 and C++ access arrays in row-major order. An analogous example in C and 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 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 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 are temporally reused with respect to some loop other than the innermost (temporal reuse is described in “Data reuse” on page 71). The unroll and jam optimization is automatically applied only to those loops that consist strictly of a basic block.

Loop unroll and jam takes place at +O3 and above and is not enabled by default in the HP compilers. To enable loop unroll and jam on the command line, use the `+Oloop_unroll_jam` option. This allows both automatic and directive-specified unroll and jam. Specifying `+Onoloop_transform` disables loop unroll and jam, loop distribution, loop interchange, loop blocking, loop fusion, and loop unroll.

The `unroll_and_jam` directive and pragma also enables this transformation. The `no_unroll_and_jam` directive and pragma is used to disable loop unroll and jam for an individual loop.



The forms of these directives and pragmas are shown in Table 9.

**Table 9** **Forms of unroll\_and\_jam, no\_unroll\_and\_jam directives and pragmas**

Language	Form
Fortran	C\$DIR UNROLL_AND_JAM[ (UNROLL_FACTOR= <i>n</i> ) ] C\$DIR NO_UNROLL_AND_JAM
C	#pragma _CNX unroll_and_jam[ (unroll_factor= <i>n</i> ) ] #pragma _CNX no_unroll_and_jam

where

unroll\_factor=*n*

allows you to specify an unroll factor for the loop in question.

**NOTE**

Because 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, is not the innermost loop. You can determine which loops in a nest are innermost by compiling the nest without any directives and examining the Optimization Report, described in “Optimization Report,” on page 151.

**Example**

**Unroll and jam**

Consider the following matrix multiply loop:

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

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

Register exploitation is vastly increased on this loop by unrolling and jamming the I and J loops. First, the compiler unrolls the I loop. To simplify the illustration, an unrolling factor of 2 for I is used. This is the number of times the contents of the loop are replicated.

## Loop and cross-module optimization features

### Loop unroll and jam

The following Fortran example shows this replication:

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
    ENDDO
  ENDDO
  DO J = 1, N
    DO K = 1, N
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

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

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

This new loop can exploit registers for two additional references:  $A(I, J)$  and  $A(I+1, J)$ . However, the compiler still has the  $J$  loop to unroll and jam. An unroll factor of 4 for the  $J$  loop is used, in which case unrolling gives the following:

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
    DO K = 1, N
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
    ENDDO
    DO K = 1, N
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
    ENDDO
    DO K = 1, N
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

Fusing (jamming) the unrolled loop results in the following:

```

DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO

```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original loop could use no more than 3 registers. This unrolled-and-jammed loop can use 14, one for each of the following references:

A(I,J)	B(I,K)	C(K,J)	A(I+1,J)
B(I+1,K)	A(I,J+1)	C(K,J+1)	A(I+1,J+1)
A(I,J+2)	C(K,J+2)	A(I,J+3)	A(I+1,J+2)
A(I+1,J+3)	C(K,J+3)		

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 `FMPYFADD` instruction, which is available with PA-8x00 processors. This instruction doubles the speed of the operations in the body of the loop by simultaneously performing related adds and multiplies.

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 select a larger unrolling factor, creating a much larger  $\kappa$  loop body. This would result in increased register exploitation and fewer loads and stores per operation.

**NOTE**

Excessive unrolling may introduce extra register spills if the unrolled and jammed loop body becomes too large. Each cache line has a 32-bit register value; register spills occur when this value is exceeded. This most often occurs as a result of continuous loop unrolling. Register spills may have negative effects on performance.

### Loop unroll and jam

You should attempt to select unroll factor values that align data references in the innermost loop on cache boundaries. As a result, references to the consecutive memory regions in the innermost loop can have very high cache hit ratios. Unroll factors of 5 or 7 may not be good choices because most array element sizes are either 4 bytes or 8 bytes and the cache line size is 32 bytes. Therefore, an unroll factor of 2 or 4 is more likely to effectively exploit cache line reuse for the references that access consecutive memory regions.

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. To increase the replication limit and possibly increase your compile time and code size, specify the `+Onosize` and `+Onolimit` compiler options.

---

## Preventing loop reordering

The `no_loop_transform` directive or pragma allows you to prevent all loop-reordering transformations on the immediately following loop.

The form of this directive and pragma are shown in Table 10.

**Table 10**

**Form of `no_loop_transform` directive and pragma**

Language	Form
Fortran	<code>C\$DIR NO_LOOP_TRANSFORM</code>
C	<code>#pragma _CNX no_loop_transform</code>

Use the command-line option `+Onolooop_transform` (at `+O3` and above) to disable loop distribution, loop blocking, loop fusion, loop interchange, loop unroll, and loop unroll and jam at the file level.

## Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop 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 are promoted, and copies of the loop are made for each test.

### Example

#### Test promotion

Consider the following Fortran loop:

```
DO I=1, 100
  DO J=1, 100
    IF(FOO .EQ. BAR) THEN
      A(I,J) = I + J
    ELSE
      A(I,J) = 0
    ENDIF
  ENDDO
ENDDO
```

Test promotion (and loop interchange) produces the following code:

```
IF(FOO .EQ. BAR) THEN
  DO J=1, 100
    DO I=1, 100
      A(I,J) = I + J
    ENDDO
  ENDDO
ELSE
  DO J=1, 100
    DO I=1, 100
      A(I,J) = 0
    ENDDO
  ENDDO
ENDIF
```

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

Each `DO` loop in Fortran and `for` loop in C and C++ whose bounds are not known at compile-time is implicitly tested to check that the loop iterates at least once. This test may be promoted, with the promotion noted in the Optimization Report. If you see unexpected promotions in the report, this implicit testing may be the cause. For more information on the Optimization Report, see “Optimization Report,” on page 151.

---

## Cross-module cloning

Cloning is the replacement of a call to a routine by a call to a clone of that routine. The clone is optimized differently than the original routine. Cloning can expose additional opportunities for optimization across multiple source files.

Cloning at `+O4` is performed across all procedures within the program, and is disabled by specifying the `+Onoinline` command-line option. This option is described on page 124.

## Global and static variable optimizations

Global and static variable optimizations look for ways to reduce the number of instructions required for accessing global and static variables (`COMMON` and `SAVE` variables in Fortran, and `extern` and `static` variables in C and C++).

The compiler normally generates two machine instructions when referencing global variables. Depending on the locality of the global variables, single machine instructions may sometimes be used to access these variables. The linker rearranges the storage location of global and static data to increase the number of variables that are referenced by single instructions.

## Global variable optimization coding standards

Because this optimization rearranges the location and data alignment of global variables, follow the programming practices given below:

- Do not make assumptions about the relative storage location of variables, such as generating a pointer by adding an offset to the address of another variable.
- Do not rely on pointer or address comparisons between two different variables.
- Do not make assumptions about the alignment of variables, such as assuming that a `short` integer is aligned the same as an integer.

## Inlining across multiple source files

Inlining substitutes function calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. This may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns. See the section "Inlining within a single source file" on page 55 for an example of inlining.

Inlining at +O4 is performed across all procedures within the program. Inlining at +O3 is done within one file.

Inlining is affected by the +O[no]inline[=*namelist*] and +Oinline\_budget=*n* command-line options. See "Controlling optimization," on page 113 for more information on these options.



## 6 Parallel optimization features

This chapter discusses parallel optimization features available with the HP-UX compilers, including those inherent in optimization levels +O3 and +O4. This includes a discussion of the following topics:

- Levels of parallelism
- Threads
- Idle thread states
- Parallel optimizations
- Inhibiting parallelization
- Reductions
- Preventing parallelization
- Parallelism in the aC++ compiler
- Cloning across multiple source files

For more information as to specific parallel command-line options, as well as pragmas and directives, please see “Controlling optimization,” on page 113.

## Levels of parallelism

In the HP compilers, parallelism exists at the loop level, task level, and region level, as described in Chapter 9, “Parallel programming techniques”. These are briefly described as follows.

- HP compilers automatically exploit loop-level parallelism. This type of parallelism involves dividing a loop into several smaller iteration spaces and scheduling these to run simultaneously on the available processors. For more information, see “Parallelizing loops” on page 178.

Using the `+Oparallel` option at `+O3` and above allows the compiler to automatically parallelize loops that are profitable to parallelize.

Only loops with iteration counts that can be determined 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.

- Specify task-level parallelism using the `begin_tasks`, `next_task` and `end_tasks` directives and pragmas, as discussed in the section “Parallelizing tasks” on page 192.
- Specify parallel regions using the `parallel` and `end_parallel` directives and pragmas, as discussed in the section “Parallelizing regions” on page 197. These directives and pragmas allow the compiler to run identified sections of code in parallel.

## Loop-level parallelism

HP compilers locate parallelism at the loop level, generating parallel code that is automatically run on as many processors as are available at runtime. Normally, these are all the processors on the same system where your program is running. You can specify a smaller number of processors using any of the following:

- `loop_parallel(max_threads=m)` directive and pragma—available in Fortran 90 and C
- `prefer_parallel(max_threads=m)` directive and pragma—available in Fortran 90 and C

For more information on the `loop_parallel` and `prefer_parallel` directives and pragmas see Chapter 9, “Parallel programming techniques”.

- `MP_NUMBER_OF_THREADS` environment variable—This variable is read at runtime by your program. If this variable is set to some positive integer  $n$ , your program executes on  $n$  processors.  $n$  must be less than or equal to the number of processors in the system where the program is executing.

## Automatic parallelization

Automatic parallelization is useful for programs containing loops. You can use compiler directives or pragmas to improve on the automatic optimizations and to assist the compiler in locating additional opportunities for parallelization.

If you are writing your program entirely under the message-passing paradigm, you must explicitly handle parallelism as discussed in the *HP MPI User's Guide*.

### Example

#### Loop-level parallelism

This example begins with the following Fortran code:

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

Assuming that the `I` loop does not contain any parallelization-inhibiting code, this program can be parallelized to run on eight 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 processor 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.

At a certain point, however, adding more processors does not improve performance. The compiler generates code that runs on as many processors as are available, but the dynamic selection optimization (described in the section “Dynamic selection” on page 102) ensures that

parallel code is executed only if it is profitable to do so. If the number of available processors does not evenly divide the number of iterations, some processors perform fewer iterations than others.

---

## Threads

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.

On a V-Class server, parallel shared-memory programs run as a collection of threads on multiple processors. When a program starts, a separate execution thread is created on each system processor on which the program is running. All but one of these threads is then idle. The nonidle thread is known as thread 1, and this thread runs all of the serial code in the program.

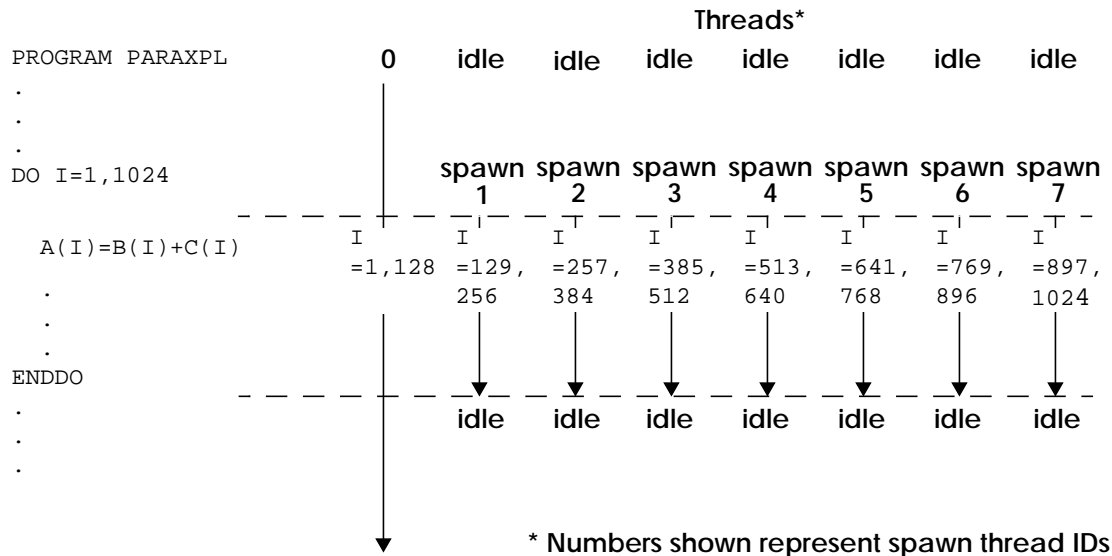
Spawn thread IDs are assigned only to nonidle threads when they are spawned. This occurs when thread 1 encounters parallelism and “wakes up” other idle threads to execute the parallel code. Spawn thread IDs are consecutive, ranging from 0 to  $N-1$ , where  $N$  is the number of threads spawned as a result of the spawn operation. This operation defines the current spawn context. The spawn context is the loop, task list, or region that initiates the spawning of the threads. Spawn thread IDs are valid only within a given spawn context.

This means that the idle threads are not assigned spawn thread IDs at the time of their creation. When thread 1 encounters a parallel loop, task, or region, 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 13.

**NOTE**

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.

**Figure 13** One-dimensional parallelism in threads



## Loop transformations

Figure 13 above shows that various loop transformations can affect the manner in which a loop is parallelized.

To implement this, the compiler transforms the loop in a manner similar to strip mining. However, unlike in strip mining, the outer loop is conceptual. Because the strips execute on different processors, there is no processor to run an outer loop like the one created in traditional strip mining.

Instead, the loop is transformed. 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.

### Example

#### Loop transformations

Consider the previous Fortran example written for an unspecified number of iterations:

```

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

```

Parallel optimization features  
Threads

The code shown in Figure 14 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 perform fewer iterations than others.

**Figure 14**      **Conceptual strip mine for parallelization**

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

$\text{NumThreads}$  is the number of available threads.  $\text{ThrdID}$  is the ID number of the thread this particular loop runs 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 Figure 13, 8 loops would be spawned, with  $\text{ThrdIDs} = 0$  through 7. These 8 loops are illustrated in Figure 15.

**Figure 15**      **Parallelized loop**

```
DO I = 1, 128
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 0**

```
DO I = 129, 256
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 1**

```
DO I = 257, 384
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 2**

```
DO I = 385, 512
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 3**

```
DO I = 513, 640
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 4**

```
DO I = 641, 768
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 5**

```
DO I = 769, 896
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 6**

```
DO I = 897, 1024
  A(I) = B(I) + C(I)
ENDDO
```

**Thread 7**

**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.

In these examples, the data being manipulated within the loop is disjoint so that 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 “Parallel synchronization,” on page 233.

---

## Idle thread states

Idle threads can be suspended or spin-waiting. Suspended threads release control of the processor while spin-waiting threads repeatedly check an encached global semaphore that indicates whether or not they have code to execute. This obviously prevents any other process from gaining control of the CPU and can severely degrade multiprocess performance.

Alternately, waking a suspended thread takes substantially longer than activating a spin-waiting thread. By default, idle threads spin-wait briefly after creation or a join, then suspend themselves if no work is received.

When threads are suspended, HP-UX may schedule threads of another process on their processors in order to balance machine load. However, threads have an affinity for their original processors. HP-UX tries to schedule unsuspended threads to their original processors in order to exploit the presence of any data encached during the thread's last timeslice. This occurs only if the original processor is available. Otherwise, the thread is assigned to the first processor to become available.

## Determining idle thread states

Use the `MP_IDLE_THREADS_WAIT` environment variable to determine how threads wait. The form of the `MP_IDLE_THREADS_WAIT` environment variable is shown in Table 11.

**Table 11**

**Form of `MP_IDLE_THREADS_WAIT` environment variable**

Language	Form
Fortran, C	<code>setenv MP_IDLE_THREADS_WAIT=<i>n</i></code>



where

$n$  is the integer value, represented in milliseconds, that the threads spin-wait. These have values as described below:

- For  $n$  less than 0, the threads spin-wait.
- For  $n$  equal to or greater than 0, the threads spin-wait for  $n$  milliseconds before being suspended.

By default, idle threads spin-wait briefly after creation or a join. They then suspend themselves if no work is received.

## Parallel optimizations

Simple loops can be parallelized without the need for extensive transformations. However, most loop transformations do enhance optimum parallelization. For instance, loop interchange orders loops so 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. It also ensures that each processor is working on nonoverlapping array data.

## Dynamic selection

The compiler has no way of determining how many processors are available to run compiled code. Therefore, it sometimes generates both serial and parallel code for loops that are parallelized. Replicating the loop in this manner is called cloning, and the resulting versions of the loop are called clones. Cloning is also performed when the loop-iteration count is unknown at compile-time.

It is not always profitable, however, to run the parallel clone when multiple processors are available. Some 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.

## Workload-based dynamic selection

HP 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. It only writes a parallel version to the executable if it is profitable to do so.

If the parallel version will not be needed, the compiler can omit it from the executable to further enhance performance. This eliminates 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. It then runs the parallel version of the loop only if it is profitable to do so.

When specified with `+Oparallel` at `+O3`, workload-based dynamic selection is enabled by default. The compiler only generates a parallel version of the loop when `+Onodynsel` is selected, thereby disabling dynamic selection. 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.

### **`dynsel`, `no_dynsel`**

The `dynsel` and `no_dynsel` directives are used to specify dynamic selection for specific loops in programs compiled using the `+Onodynsel` option or to provide trip count information for specific loops in programs compiled with dynamic selection enabled.

To disable dynamic selection for selected loops by using the `no_dynsel` compiler directive or pragma. This directive or pragma is used to disable dynamic selection on specific loops in programs compiled with dynamic selection enabled.

The form of these directives and pragmas are shown in Table 12.

**Table 12**      **Form of `dynsel` directive and pragma**

<b>Language</b>	<b>Form</b>
Fortran	<code>C\$DIR DYNSEL [(THREAD_TRIP_COUNT = <i>n</i>)]</code> <code>C\$DIR NO_DYNSEL</code>
C	<code>#pragma _CNX dynsel [(thread_trip_count = <i>n</i>)]</code> <code>#pragma _CNX no_dynsel</code>

where

`thread_trip_count`

is an optional attribute used to specify threshold iteration counts.

When `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 a trip count is not specified for a `dynsel` directive or `pragma`, the compiler uses a heuristic to estimate the actual execution costs. This estimate is then used to determine if it is profitable to execute the loop in parallel.

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 code sizes. To increase the replication limit (and possibly increase your compile time and code size), specify the `+Onosize` `+Onolimit` compiler options. These are described in “Controlling optimization,” on page 113.

---

## Inhibiting parallelization

Certain constructs, such as loop-carried dependences, inhibit parallelization. Other types of constructs, such as procedure calls and I/O statements, inhibit parallelism for the same reason they inhibit localization. An exception to this is that more categories of loop-carried dependences can inhibit parallelization than data localization. This is described in the following sections.

### Loop-carried dependences (LCDs)

The specific loop-carried dependences (LCDs) that inhibit data localization represent a very small portion of all loop-carried dependences. A much broader set of LCDs inhibits parallelization. Examples of various parallel-inhibiting LCDs follows.

#### Example

##### Parallel-inhibiting LCDs

One type of LCD exists when one iteration references a variable whose value is assigned on a later iteration. The Fortran loop below contains this type of 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  $I+1$ th 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,  $A(I+1)$  has usually already been computed by another processor. It is, in fact, the first value computed by another processor. Because the calculation depends on  $A(I+1)$  not being computed yet, this would produce wrong answers.

#### Example

##### Parallel-inhibiting LCDs

Another type of 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.

## Parallel optimization features

### Inhibiting parallelization

```
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 result.

Backward LCDs inhibit parallelism because if the loop is broken up to run on several processors,  $A(I-1)$  are not computed for the first iteration of the loop on every processor except the processor running the chunk of the loop containing  $I = 1$ .

### Example

#### Output LCDs

An output LCD exists when the same memory location is assigned values on two or more iterations. A potential 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  $A$  is 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. This cannot be guaranteed if the loop is run in parallel.

### Example

#### Apparent LCDs

The compiler chooses to 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 section “Loop-carried dependences (LCDs)” on page 59.

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  $\mathcal{J}$  are identical, so there is no output LCD and the loop is safe to parallelize. If any of the  $\mathcal{J}$  values are identical, wrong answers could result.

---

## Reductions

In many cases, the compiler can recognize and parallelize loops containing a special class of dependence 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.`, or `.XOR.`

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.

Generally, the compiler automatically recognizes reductions in a loop and is able to parallelize the loop. If the loop is under the influence of the `prefer_parallel` directive or pragma, the compiler still recognizes reductions.

However, in a loop being manipulated by the `loop_parallel` directive or pragma, reduction analysis is not performed. Consequently, the loop may not be correctly parallelized unless the reduction is enforced using the `reduction` directive or pragma.

The form of this directive and pragma is shown in Table 13.

**Table 13** Form of reduction directive and pragma

Language	Form
Fortran	<code>C\$DIR REDUCTION</code>
C	<code>#pragma _CNX reduction</code>

**Example** Reduction



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-specific copies of `ASUM` for each thread that runs the loop. When each parallel thread completes its portion of the loop, thread 0 for the current spawn context accumulates the thread-specific values into the global `ASUM`.

The following Fortran example shows the use of the `reduction` directive on the above code. `loop_parallel` is described on page 179. `loop_private` is described on page 210.

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(FUNCTEMP), REDUCTION(SUM)
DO I = 1, N
  .
  .
  .
  FUNCTEMP = FUNC(X(I))
  SUM = SUM + FUNCTEMP
  .
  .
  .
ENDDO
```

---

## Preventing parallelization

You can prevent parallelization on a loop-by-loop basis using the `no_parallel` directive or pragma. The form of this directive and pragma is shown in Table 14.

**Table 14** Form of `no_parallel` directive and pragma

Language	Form
Fortran	<code>C\$DIR NO_PARALLEL</code>
C	<code>#pragma _CNX no_parallel</code>

Use these directives to prevent parallelization of the loop that immediately follows them. Only parallelization is inhibited; all other loop optimizations are still applied.

### Example

`no_parallel`

The following Fortran example illustrates the use of `no_parallel`:

```
      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.

The `+Onoautopar` compiler option is available to disable automatic parallelization but allows parallelization of directive-specified loops. Refer to “Controlling optimization,” on page 113, and “Parallel programming techniques,” on page 175, for more information on `+Onoautopar`.

## Parallelism in the aC++ compiler

Parallelism in the aC++ compiler is available through the use of the following command-line options or libraries:

- `+O3 +Oparallel` or `+O4 +Oparallel` optimization options— Automatic parallelization is available from the compiler; see the section “Levels of parallelism” on page 94 for more information.
- HP MPI—HP’s implementation of the message-passing interface; see the *HP MPI User’s Guide* for more information.
- Pthreads (POSIX threads)— See the `pthread(3t)` man page or the manual *Programming with Threads on HP-UX* for more information.

None of the pragmas described in this book are currently available in the HP aC++ compiler. However, aC++ does support the memory classes briefly explained in “Controlling optimization,” on page 113, and more specifically in “Memory classes,” on page 223. These classes are implemented through the storage class specifiers `node_private` and `thread_private`.

---

## Cloning across multiple source files

Cloning is the replacement of a call to a routine by a call to a clone of that routine. The clone is optimized differently than the original routine. Cloning can expose additional opportunities for interprocedural optimization.

Cloning at +O4 is performed across all procedures within the program. Cloning at +O3 is done within one file. Cloning is enabled by default. It is disabled by specifying the `+Onoinline` command-line option.

# 7 **Controlling optimization**

The HP-UX compiler set includes a group of optimization controls that are used to improve code performance. These controls can be invoked from either the command line or from within a program using certain directives and pragmas.

This chapter includes a discussion of the following topics:

- Command-line optimization options
- Invoking command-line options
- C aliasing options
- Optimization directives and pragmas

Refer to Chapter 3, “Optimization levels” for information on coding guidelines that assist the optimizer. See the `f90(1)`, `cc(1)`, and `aCC(1)` man pages for information on compiler options in general.

**NOTE**

The HP aC++ compiler does not support the pragmas described in this chapter.

## Command-line optimization options

This section lists the command-line optimization options available for use with the HP C, C++, and Fortran 90 compilers. Table 15 describes the options and the optimization levels at which they are used.

**Table 15** **Command-line optimization options**

Optimization options	Valid optimization levels
<b>Command-line options</b>	
+O[no]aggressive	+02, +03, +04
+O[no]all	all
+O[no]autopar (must be used with the +Oparallel option at +03 or above)	+03, +04
+O[no]conservative	+02, +03, +04
+O[no]dataprefetch	+02, +03, +04
+O[no]dynsel (must be used with the +Oparallel option at +03 or above)	+03, +04
+O[no]entrsched	+01, +02, +03, +04
+O[no]fail_safe	+01, +02, +03, +04
+O[no]fastaccess	all
+O[no]fltacc	+02, +03, +04
+O[no]global_ptrs_unique[=namelist] (C only)	+02, +03, +04
+O[no]info	all

Optimization options	Valid optimization levels
+O[no]initcheck	+O2, +O3, +O4
+O[no]inline[= <i>namelist</i> ]	+O3, +O4
+Oinline_budget= <i>n</i>	+O3, +O4
+O[no]libcalls	all
+O[no]limit	+O2, +O3, +O4
+O[no]loop_block	+O3, +O4
+O[no]loop_transform	+O3, +O4
+O[no]loop_unroll[= <i>unroll_factor</i> ]	+O2, +O3, +O4
+O[no]loop_unroll_jam	+O3, +O4
+O[no]moveflops	+O2, +O3, +O4
+O[no]multiprocessor	+O2, +O3, +O4
+O[no]parallel	+O3, +O4
+O[no]parmsoverlap	+O2, +O3, +O4
+O[no]pipeline	+O2, +O3, +O4
+O[no]procelim	all
+O[no]ptrs_ansi	+O2, +O3, +O4
+O[no]ptrs_strongly_typed	+O2, +O3, +O4
+O[no]ptrs_to_globals[= <i>namelist</i> ] (C only)	+O2, +O3, +O4
+O[no]regreassoc	+O2, +O3, +O4
+O[no]report[= <i>report_type</i> ]	+O3, +O4
+O[no]sharedgra	+O2, +O3, +O4

Controlling optimization  
Command-line optimization options

<b>Optimization options</b>	<b>Valid optimization levels</b>
+O[no]signedpointers (C/C++ only)	+O2, +O3, +O4
+O[no]size	+O2, +O3, +O4
+O[no]static_prediction	<b>all</b>
+O[no]vectorize	+O3, +O4
+O[no]volatile	+O1, +O2, +O3, +O4
+O[no]whole_program_mode	+O4



## Invoking command-line options

At each optimization level, you can turn specific optimizations on or off using the `+O[no] optimization` option. The *optimization* parameter is the name of a specific optimization. The optional prefix `[no]` disables the specified optimization.

The following sections describe the optimizations that are turned on or off, their defaults, and the optimization levels at which they may be used. In syntax descriptions, *namelist* represents a comma-separated list of names.

### **+O[no]aggressive**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onoaggressive

`+O[no]aggressive` enables or disables optimizations that can result in significant performance improvement, and can change a program's behavior. This includes the optimizations invoked by the following advanced options (these are discussed separately in this chapter):

- `+Osignedpointers` (C and C++)
- `+Oentrysched`
- `+Onofltacc`
- `+Olibcalls`
- `+Onoinitcheck`
- `+Ovectorize`

## **+O[no]all**

**Optimization level:** all

**Default:** +Onoall

**Equivalent option:** +Oall option is equivalent to specifying +O4  
+Oaggressive +Onolimit

+Oall performs maximum optimization, including aggressive optimizations and optimizations that can significantly increase compile time and memory usage.

## **+O[no]autopar**

**Optimization level:** +O3, +O4 (+Oparallel must also be specified)

**Default:** +Oautopar

When used with +Oparallel option, +Oautopar causes the compiler to automatically parallelize loops that are safe to parallelize. A loop is considered safe to parallelize if its iteration count can be determined at runtime before loop invocation. It must also contain no loop-carried dependences, procedure calls, or I/O operations.

A loop-carried dependence exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration.

When used with +Oparallel, the +Onoautopar option causes the compiler to parallelize only those loops marked by the `loop_parallel` or `prefer_parallel` directives or pragmas. Because the compiler does not automatically find parallel tasks or regions, user-specified task and region parallelization is not affected by this option.

C pragmas and Fortran directives are used to improve the effect of automatic optimizations and to assist the compiler in locating additional opportunities for parallelization. See “Optimization directives and pragmas” on page 146 for more information.

## **+O[no]conservative**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onoconservative

**Equivalent option:** +Oconservative is equivalent to +Onoaggressive

+O[no]conservative causes the optimizer to make or not make conservative assumptions about the code when optimizing. +Oconservative is useful in assuming a particular program's coding style, such as whether it is standard-compliant. Specifying +Onoconservative disables any optimizations that assume standard-compliant code.

## **+O[no]dataprefetch**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onodataprefetch

When +Odataprefetch is used, the optimizer inserts instructions within innermost loops to explicitly prefetch data from memory into the data cache. For cache lines containing data to be written, +Odataprefetch prefetches the cache lines so that they are valid for both read and write access. Data prefetch instructions are inserted only for data referenced within innermost loops using simple loop-varying addresses in a simple arithmetic progression. It is only available for PA-RISC 2.0 targets.

The math library `libm` contains special prefetching versions of vector routines. If you have a PA-RISC 2.0 application containing operations on arrays larger than one megabyte in size, using +Ovectorize in conjunction with +Odataprefetch may substantially improve performance.

You can also use the +Odataprefetch option for applications that have high data cache miss overhead.

## +O[no]dynsel

**Optimization level:** +O3, +O4 (+Oparallel must also be specified)

**Default:** +Odynsel

When specified with +Oparallel, +Odynsel enables workload-based dynamic selection. For parallelizable loops whose iteration counts are known at compile time, +Odynsel causes the compiler to generate either a parallel or a serial version of the loop—depending on which is more profitable.

This optimization also causes the compiler to generate both 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.

The +Onodynsel option disables dynamic selection and tells the compiler that it is profitable to parallelize all parallelizable loops. The dynsel directive and pragma are used to enable dynamic selection for specific loops, when +Onodynsel is in effect. See the section “Dynamic selection” on page 102 for additional information.

## +O[no]entrsched

**Optimization level:** +O1, +O2, +O3, +O4

**Default:** +Onoentrsched

+Oentrsched optimizes instruction scheduling on a procedure's entry and exit sequences by unwinding in the entry and exit regions. Subsequently, this option is used to increase the speed of an application.

+O[no]entrsched can also change the behavior of programs performing exception-handling or that handle asynchronous interrupts. The behavior of setjmp() and longjmp() is not affected.

## **+O[no]fail\_safe**

**Optimization level:** +O1, +O2, +O3, +O4

**Default:** +Ofail\_safe

+Ofail\_safe allows your compilations to continue when internal optimization errors are detected. When an error is encountered, this option issues a warning message and restarts the compilation at +O0. The +Ofail\_safe option is disabled when you specify +Oparallel with +O3 or +O4 to compile with parallelization.

Using +Onofail\_safe aborts your compilation when internal optimization errors are detected.

## **+O[no]fastaccess**

**Optimization level:** +O0, +O1, +O2, +O3, +O4

**Default:** +Onofastaccess at +O0, +O1, +O2 and +O3;  
+Ofastaccess at +O4

+Ofastaccess performs optimization for fast access to global data items. Use +Ofastaccess to improve execution speed at the expense of longer compile times.

## **+O[no]fltacc**

**Optimization level:** +O2, +O3, +O4

**Default:** none (See Table 16.)

+O[no]fltacc enables or disables optimizations that cause imprecise floating-point results.

+Ofltacc disables optimizations that cause imprecise floating-point results. Specifying +Ofltacc disables the generation of Fused Multiply-Add (FMA) instructions, as well as other floating-point optimizations. Use +Ofltacc if it is important that the compiler evaluates floating-point expressions according to the order specified by the language standard.

+Onofltacc improves execution speed at the expense of floating-point precision. The +Onofltacc option allows the compiler to perform floating-point optimizations that are algebraically correct, but may

Controlling optimization  
Invoking command-line options

result in numerical differences. These differences are generally insignificant. The `+Onofltacc` option also enables the optimizer to generate FMA instructions.

If you optimize code at `+O2` or higher, and do not specify `+Onofltacc` or `+Ofltacc`, the optimizer uses FMA instructions. However, it does not perform floating-point optimizations that involve expression reordering. FMA is implemented by the PA-8x00 instructions `FMPYFADD` and `FMPYNFADD` and improves performance. Occasionally, these instructions may produce results that may differ in accuracy from results produced by code without FMA. In general, the differences are slight.

Table 16 presents a summary of the preceding information.

**Table 16** `+O[no]f1tacc` and floating-point optimizations

Option specified <sup>a</sup>	FMA optimizations	Other floating-point optimizations
<code>+Of1tacc</code>	Disabled	Disabled
<code>+Onof1tacc</code>	Enabled	Enabled
neither option is specified	Enabled	Disabled

a. `+O[no]f1tacc` is only available at `+O2` and above.

**`+O[no]global_ptrs_unique[=namelist]`**

**Optimization level:** `+O2`, `+O3`, `+O4`

**Default:** `+Onoglobal_ptrs_unique`

**NOTE**

This option is not available in Fortran or C++.

Using this C compiler option identifies unique global pointers so that the optimizer can generate more efficient code in the presence of unique pointers, such as using copy propagation and common subexpression elimination. A global pointer is unique if it does not alias with any variable in the entire program.

This option supports a comma-separated list of unique global pointer variable names, represented by *namelist* in `+O[no]global_ptrs_unique[=namelist]`. If *namelist* is not specified, using `+O[no]global_ptrs_unique` informs the compiler that all [no] global pointers are unique.

The example below states that no global pointers are unique, except a and b:

```
+Oglobal_ptrs_unique=a,b
```

The next example says that all global pointers are unique except a and b:

```
+Onoglobal_ptrs_unique=a,b
```

### **+O[no]info**

**Optimization level:** +O0, +O1, +O2, +O3, +O4

**Default:** +Onoinfo

+Oinfo displays informational messages about the optimization process. This option is used at all optimization levels, but is most useful at +O3 and +O4. For more information about this option, see Chapter 8, “Optimization Report” on page 113.

### **+O[no]initcheck**

**Optimization level:** +O2, +O3, +O4

**Default:** unspecified

+O[no]initcheck performs an initialization check for the optimizer. The optimizer has three possible states that check for initialization: on, off, or unspecified.

- When on (+Oinitcheck), the optimizer initializes to zero any local, scalar, and nonstatic variables that are uninitialized with respect to at least one path leading to a use of the variable.
- When off (+Onoinitcheck), the optimizer issues warning messages when it discovers definitely uninitialized variables, but does not initialize them.
- When unspecified, the optimizer initializes to zero any local, scalar, nonstatic variables that are definitely uninitialized with respect to all paths leading to a use of the variable.

## **+O[no]inline[=*namelist*]**

**Optimization level:** +O3, +O4

**Default:** +Oinline

When +Oinline is specified without a name list, any function can be inlined. For successful inlining, follow the prototype definitions for function calls in the appropriate header files.

When specified with a name list, the named functions are important candidates for inlining. For example, the following statement indicates that inlining be strongly considered for `foo` and `bar`:

```
+Oinline=foo,bar +Onoinline
```

All other routines are not considered for inlining because +Onoinline is given.

### NOTE

The Fortran 90 and aC++ compilers accept only +O[no]inline. No *namelist* values are accepted.

Use the +Onoinline[=*namelist*] option to exercise precise control over which subprograms are inlined. Use of this option is guided by knowledge of the frequency with which certain routines are called and may be warranted by code size concerns.

When this option is disabled with a name list, the compiler does not consider the specified routines as candidates for inlining. For example, the following statement indicates that inlining should not be considered for `baz` and `x`:

```
+Onoinline=baz,x
```

All other routines are considered for inlining because +Oinline is the default.



## **+Oinline\_budget=*n***

**Optimization level:** +O3, +O4

**Default:** +Oinline\_budget=100

In +Oinline\_budget=*n*, *n* is an integer in the range 1 to 1000000 that specifies the level of aggressiveness, as follows:

*n* = 100            Default level of inlining

*n* > 100           More aggressive inlining

The optimizer is less restricted by compilation time and code size when searching for eligible routines to inline

*n* = 1              Only inline if it reduces code size

The +Onolimit and +Osize options also affect inlining. Specifying the +Onolimit option implies specifying +Oinline\_budget=200. The +Osize option implies +Oinline\_budget=1. However, +Oinline\_budget takes precedence over both of these options. This means that you can override the effects on inlining of the +Onolimit and +Osize options, by specifying the +Oinline\_budget option on the same compile line.

## **+O[no]libcalls**

**Optimization level:** +O0, +O1, +O2, +O3, +O4

**Default:** +Onolibcalls at +O0 and +O1;  
+Olibcalls at +O2, +O3, and +O4

+Olibcalls increases the runtime performance of code that calls standard library routines in simple contexts. The +Olibcalls option expands the following library calls inline:

- strcpy()
- sqrt()
- fabs()
- alloca()

Inlining takes place only if the function call follows the prototype definition in the appropriate header file. A single call to printf() may be replaced by a series of calls to putchar(). Calls to sprintf() and

Controlling optimization  
Invoking command-line options

`strlen()` may be optimized more effectively, including elimination of some calls producing unused results. Calls to `setjmp()` and `longjmp()` may be replaced by their equivalents `_setjmp()` and `_longjmp()`, which do not manipulate the process's signal mask.

Using the `+Olibcalls` option invokes millicode versions of frequently called math functions. Currently, there are millicode versions for the following functions:

<code>acos</code>	<code>asin</code>	<code>atan</code>	<code>atan2</code>
<code>cos</code>	<code>exp</code>	<code>log</code>	<code>log10</code>
<code>pow</code>	<code>sin</code>	<code>tan</code>	

See the *HP-UX Floating-Point Guide* for the most up-to-date listing of the math library functions.

`+Olibcalls` also improves the performance of selected library routines (when you are not performing error checking for these routines). The calling code must not expect to access `ERRNO` after the function's return.

Using `+Olibcalls` with `+Ofltacc` gives different floating-point calculation results than those given using `+Olibcalls` without `+Ofltacc`.

## **+O[no]limit**

**Optimization level:** `+O2`, `+O3`, `+O4`

**Default:** `+Olimit`

The `+Olimit` option suppresses optimizations that significantly increase compile-time or that can consume a considerable amount of memory.

The `+Onolimit` option allows optimizations to be performed, regardless of their effects on compile-time and memory usage. Specifying the `+Onolimit` option implies specifying `+Oinline_budget=200`. See the section "`+Oinline_budget=n`" on page 125 for more information.

## +O[no]loop\_block

**Optimization level:** +O3, +O4

**Default:** +Onoloop\_block

+O[no]loop\_block enables or disables blocking of eligible loops for improved cache performance. The +Onoloop\_block option disables both automatic and directive-specified loop blocking. For more information on loop blocking, see the section “Loop blocking” on page 70.

## +O[no]loop\_transform

**Optimization level:** +O3, +O4

**Default:** +Oloop\_transform

+O[no]loop\_transform enables or disables transformation of eligible loops for improved cache performance. The most important transformation is the interchange of nested loops to make the inner loop unit stride, resulting in fewer cache misses.

The other transformations affected by +O[no]loop\_transform are loop distribution, loop blocking, loop fusion, loop unroll, and loop unroll and jam. See “Optimization levels,” on page 25 for information on loop transformations.

If you experience any problem while using +Oparallel, +Onoloop\_transform may be a helpful option.

## +O[no]loop\_unroll[=*unroll factor*]

**Optimization level:** +O2, +O3, +O4

**Default:** +Oloop\_unroll=4

+Oloop\_unroll enables loop unrolling. When you use +Oloop\_unroll, you can also suggest the unroll factor to control the code expansion. The default unroll factor is four, meaning that the loop body is replicated four times. By experimenting with different factors, you may improve the performance of your program. In some cases, the compiler uses its own unroll factor.

[Controlling optimization](#)  
[Invoking command-line options](#)

The `+Onolloop_unroll` option disables partial and complete unrolling. Loop unrolling improves efficiency by eliminating loop overhead, and can create opportunities for other optimizations, such as improved register use and more efficient scheduling. See the section “Loop unrolling” on page 45 for more information on unrolling.

### **+O[no]loop\_unroll\_jam**

**Optimization level:** +O3, +O4

**Default:** +Onolloop\_unroll\_jam

The `+O[no]loop_unroll_jam` option enables or disables loop unrolling and jamming. The `+Onolloop_unroll_jam` option (the default) disables both automatic and directive-specified unroll and jam. Loop unrolling and jamming increases register exploitation. For more information on the unroll and jam optimization, see the section “Loop unroll and jam” on page 84.

### **+O[no]moveflops**

**Optimization level:** +O2, +O3, +O4

**Default:** +Omoveflops

`+O[no]moveflops` allows or disallows moving conditional floating-point instructions out of loops. The behavior of floating-point exception handling may be altered by this option.

Use `+Onomoveflops` if floating-point traps are enabled and you do not want the behavior of floating-point exceptions to be altered by the relocation of floating-point instructions.

## **+O[no]multiprocessor**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onomultiprocessor

Specifying the +Omultiprocessor option at +O2 and above tells the compiler to appropriately optimize several different processes on multiprocessor machines. The optimizations are those appropriate for executables and shared libraries.

Enabling this option incorrectly (such as on a uniprocessor machine) may cause performance problems.

Specifying +Onomultiprocessor (the default) disables the optimization of more than one process running on multiprocessor machines.

## **+O[no]parallel**

**Optimization level:** +O3, +O4

**Default:** +Onoparallel

The +Onoparallel option is the default for all optimization levels. This option disables automatic and directive-specified parallelization.

If you compile one or more files in an application using +Oparallel, then the application must be linked (using the compiler driver) with the +Oparallel option to link in the proper start-up files and runtime support.

The +Oparallel option causes the compiler to:

- Recognize the directives and pragmas that involve parallelism, such as `begin_tasks`, `loop_parallel`, and `prefer_parallel`
- Look for opportunities for parallel execution in loops

The following methods are used to specify the number of processors used in executing your parallel programs:

- `loop_parallel(max_threads=m)` directive and pragma
- `prefer_parallel(max_threads=m)` directive and pragma

Controlling optimization  
Invoking command-line options

For a description of these directives and pragmas, see “Parallel programming techniques,” on page 175 and “Parallel synchronization,” on page 233. These pragmas are not available in the HP aC++ compiler.

- `MP_NUMBER_OF_THREADS` environment variable, which is read at runtime by your program. If this variable is set to some positive integer  $n$ , your program executes on  $n$  processors.  $n$  must be less than or equal to the number of processors in the system where the program is executing.

The `+Oparallel` option is valid only at optimization level `+O3` and above. For information on parallelization, see the section “Levels of parallelism” on page 94.

Using the `+Oparallel` option disables `+Ofail_safe`, which is enabled by default. See the section “`+O[no]fail_safe`” on page 121 for more information.

### **+O[no]parmsoverlap**

**Optimization level:** `+O2`, `+O3`, `+O4`

**Default (Fortran):** `+Onoparmsoverlap`

**Default (C/C++):** `+Oparmsoverlap`

`+Oparmsoverlap` causes the optimizer to assume that the actual arguments of function calls overlap in memory.

### **+O[no]pipeline**

**Optimization level:** `+O2`, `+O3`, `+O4`

**Default:** `+Opipeline`

`+O[no]pipeline` enables or disables software pipelining. If program size is more important than execution speed, use `+Onopipeline`.

Software pipelining is particularly useful for loops containing arithmetic operations on `REAL` or `REAL*8` variables in Fortran or on `float` or `double` variables in C and C++.

## **+O[no]procelim**

**Optimization level:** +O0, +O1, +O2, +O3, +O4

**Default:** +Onoprocelim at +O0, +O1, +O2, +O3;  
+Oprocelim at +O4

When +Oprocelim is specified, procedures not referenced by the application are eliminated from the output executable file. The +Oprocelim option reduces the size of the executable file, especially when optimizing at +O3 and +O4, at which inlining may have removed all of the calls to some routines.

When +Onoprocelim is specified, procedures not referenced by the application are not eliminated from the output executable file.

If the +Oall option is enabled, the +Oprocelim option is enabled.

## **+O[no]ptrs\_ansi**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onoptrs\_ansi

The +Optrs\_ansi option makes the following two assumptions, which the more aggressive +Optrs\_strongly\_typed does not:

- `int *p` is assumed to point to an `int` field of a struct or union.
- `char *` is assumed to point to any type of object.

### NOTE

This option is not available in C++.

When both +Optrs\_ansi and +Optrs\_strongly\_typed are specified, +Optrs\_ansi takes precedence.

[Controlling optimization](#)  
[Invoking command-line options](#)

## **+O[no]ptrs\_strongly\_typed**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onoptrs\_strongly\_typed

Use the C compiler option `+Optrs_strongly_typed` when pointers are type-safe. The optimizer can use this information to generate more efficient code.

**NOTE** This option is not available in C++.

Type-safe (strongly-typed) pointers point to a specific type that, in turn, only point to objects of that type. For example, a pointer declared as a pointer to an `int` is considered type-safe if that pointer points to an object of type `int` only.

Based on the type-safe concept, a set of groups are built based on object types. A given group includes all the objects of the same type.

In type-inferred aliasing, any pointer of a type in a given group (of objects of the same type) can only point to any object from the same group. It cannot point to a typed object from any other group.

Type casting to a different type violates type-inferring aliasing rules. Dynamic casting is, however, allowed, as shown in Example 41.

### **Example**

#### **Data type interaction**

The optimizer generally spills all global data from registers to memory before any modification to global variables or any loads through pointers. However, the optimizer can generate more efficient code if it knows how various data types interact.



Consider the following example (line numbers are provided for reference):

```

1 int *p;
2 float *q;
3 int a,b,c;
4 float d,e,f;
5 foo()
6 {
7     for (i=1;i<10;i++) {
8         d=e;
9         *p=...;
10        e=d+f;
11        f=*q;
12    }
13 }
```

With `+Onoptrs_strongly_typed` turned on, the pointers `p` and `q` are assumed to be disjoint because the types they point to are different types. Without type-inferred aliasing, `*p` is assumed to invalidate all the definitions. So, the use of `d` and `f` on line 10 have to be loaded from memory. With type-inferred aliasing, the optimizer can propagate the copy of `d` and `f`, thus avoiding two loads and two stores.

This option is used for any application involving the use of pointers, where those pointers are type safe. To specify when a subset of types are type-safe, use the `ptrs_strongly_typed` pragma. The compiler issues warnings for any incompatible pointer assignments that may violate the type-inferred aliasing rules discussed in the section “C aliasing options” on page 143.

## Example

### Unsafe type cast

Any type cast to a different type violates type-inferred aliasing rules. Do not use `+Optrs_strongly_typed` with code that has these “unsafe” type casts. Use the `no_ptrs_strongly_typed` pragma to prevent the application of type-inferred aliasing to the unsafe type casts.

```

struct foo{
    int a;
    int b;
} *p;

struct bar {
    float a;
    int b;
    float c;
} *q;

P = (struct foo *) q;
/* Incompatible pointer assignment
through type cast */
```

## Example

### Generally applying type aliasing

Dynamic casting is allowed with `+Optrs_strongly_typed` or `+Optrs_ansi`. A pointer dereference is called a dynamic cast if a cast is applied on the pointer to a different type.

In the example below, type-inferred aliasing is generally applied on `P`, not just to the particular dereference. Type-aliasing is applied to any other dereferences of `P`.

```
struct s {
    short int a;
    short int b;
    int c;
} *P
* (int *)P = 0;
```

For more information about type aliasing, see the section “C aliasing options” on page 143.

## +O[no]ptrs\_to\_globals[=*namelist*]

**Optimization level:** +O2, +O3, +O4

**Default:** +Optrs\_to\_globals

By default, global variables are conservatively assumed to be modified anywhere in the program. Use the C compiler option +Onoptrs\_to\_globals to specify which global variables are not modified through pointers. This allows the optimizer to make the program run more efficiently by incorporating copy propagation and common subexpression elimination.

### NOTE

This option is not available in C++.

This option is used to specify all global variables that are not modified using pointers, or to specify a comma-separated list of global variables that are not modified using pointers.

The on state for this option disables some optimizations, such as aggressive optimizations on the program's global symbols.

For example, use the command-line option

+Onoptrs\_to\_globals=a,b,c to specify global variables a, b, and c to not be accessible through pointers. The result (shown below) is that no pointer can access these global variables. The optimizer performs copy propagation and constant folding because storing to \*p does not modify a or b.

```
int a, b, c;
float *p;
foo()
{
    a = 10;
    b = 20;
    *p = 1.0;
    c = a + b;
}
```

If all global variables are unique, use the +Onoptrs\_to\_globals option without listing the global variables (that is, without using *namelist*).

Controlling optimization  
Invoking command-line options

In the example below, the address of `b` is taken. This means `b` is accessed indirectly through the pointer. You can still use `+Onoptrs_to_globals` as:

```
+Onoptrs_to_globals +Optrs_to_globals=b.  
int b,c;  
int *p  
p=&b;  
foo()
```

For more information about type aliasing, see the section “C aliasing options” on page 143.

### **+O[no]regreassoc**

**Optimization level:** +O2, +O3, +O4

**Default:** +Oregreassoc

`+O[no]regreassoc` enables or disables register reassociation. This is a technique for folding and eliminating integer arithmetic operations within loops, especially those used for array address computations.

This optimization provides a code-improving transformation supplementing loop-invariant code motion and strength reduction. Additionally, when performed in conjunction with software pipelining, register reassociation can also yield significant performance improvement.

**+O[no]report [=report\_type]**

**Optimization level:** +O3, +O4

**Default:** +Onoreport

+Oreport [=report\_type] specifies the contents of the Optimization Report. Values of *report\_type* and the Optimization Reports they produce are shown in Table 17.

**Table 17**

**Optimization Report contents**

<i>report_type</i> value	Report contents
all	Loop Report and Privatization Table
loop	Loop Report
private	Loop Report and Privatization Table
<i>report_type</i> not given (default)	Loop Report

The Loop Report gives information on optimizations performed on loops and calls. Using +Oreport (without =*report\_type*) also produces the Loop Report.

The Privatization Table provides information on loop variables that are privatized by the compiler.

+Oreport [=report\_type] is active only at +O3 and above.

The +Onoreport option does not accept any of the *report\_type* values.

For more information about the Optimization Report, see “Optimization Report,” on page 151.

+Oinfo also displays information on the various optimizations being performed by the compilers. +Oinfo is used at any optimization level, but is most useful at +O3 and above. The default at all optimization levels is +Onoinfo.

[Controlling optimization](#)  
[Invoking command-line options](#)

## **+O[no]sharedgra**

**Optimization level:** +O2, +O3, +O4

**Default:** +Osharedgra

The +Onosharedgra option disables global register allocation for shared-memory variables that are visible to multiple threads. This option may help if a variable shared among parallel threads is causing wrong answers. See the section “Global register allocation (GRA)” on page 43 for more information.

Global register allocation (+Osharedgra) is enabled by default at optimization level +O2 and higher.

## **+O[no]signedpointers**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onosignedpointers

NOTE

This option is not available in the HP Fortran 90 compiler.

The C and C++ option +O[no]signedpointers requests that the compiler perform or not perform optimizations related to treating pointers as signed quantities. This helps improve application runtime speed. Applications that allocate shared memory and that compare a pointer to shared memory with a pointer to private memory may run incorrectly if this optimization is enabled.

## **+O[no]size**

**Optimization level:** +O2, +O3, +O4

**Default:** +Onosize

The +Osize option suppresses optimizations that significantly increase code size. Specifying +Osize implies specifying +Oinline\_budget=1. See the section “+Oinline\_budget=*n*” on page 125 for additional information.

The +Onosize option does not prevent optimizations that can increase code size.

## **+O[no]static\_prediction**

**Optimization level:** +O0, +O1, +O2, +O3, +O4

**Default:** +Onostatic\_prediction

+Ostatic\_prediction turns on static branch prediction for PA-RISC 2.0 targets. Use +Ostatic\_prediction to better optimize large programs with poor instruction locality, such as operating system and database code.

PA-RISC 2.0 predicts the direction conditional branches go in one of two ways:

- Dynamic branch prediction uses a hardware history mechanism to predict future executions of a branch from its last three executions. It is transparent and quite effective, unless the hardware buffers involved are overwhelmed by a large program with poor locality.
- Static branch prediction, when enabled, predicts each branch based on implicit hints encoded in the branch instruction itself. The static branch prediction is responsible for handling large codes with poor locality for which the small dynamic hardware facility proves inadequate.

## **+O[no]vectorize**

**Optimization level:** +O3, +O4

**Default:** +Onovectorize

+Ovectorize allows the compiler to replace certain loops with calls to vector routines. Use +Ovectorize to increase the execution speed of loops.

### NOTE

This option is not available in the HP aC++ compiler.

When +Onovectorize is specified, loops are not replaced with calls to vector routines.

Because the +Ovectorize option may change the order of floating-point operations in an application, it may also change the results of those operations slightly. See the *HP-UX Floating-Point Guide* for more information.

[Controlling optimization](#)  
[Invoking command-line options](#)

The math library contains special prefetching versions of vector routines. If you have a PA2.0 application containing operations on large arrays (larger than 1 Megabyte in size), using `+Ovectorize` in conjunction with `+Odataprefetch` may improve performance.

`+Ovectorize` is also included as part of the `+Oaggressive` and `+Oall` options.

### **+O[no]volatile**

**Optimization level:** +O1, +O2, +O3, +O4

**Default:** +Onovolatile

**NOTE**

This option is not available in the HP Fortran 90 compiler.

The C and C++ option `+Ovolatile` implies that memory references to global variables cannot be removed during optimization.

The `+Onovolatile` option indicates that all globals are not of volatile class. This means that references to global variables are removed during optimization.

Use this option to control the volatile semantics for all global variables.

### **+O[no]whole\_program\_mode**

**Optimization level:** +O4

**Default:** +Onowhole\_program\_mode

Use `+Owhole_program_mode` to increase performance speed. This should be used only when you are certain that only the files compiled with `+Owhole_program_mode` directly access any globals that are defined in these files.

**NOTE**

This option is not available in the HP Fortran 90 or aC++ compilers.

`+Owhole_program_mode` enables the assertion that only the files that are compiled with this option directly reference any global variables and procedures that are defined in these files. In other words, this option asserts that there are no unseen accesses to the globals.

When this assertion is in effect, the optimizer can hold global variables in registers longer and delete inlined or cloned global procedures.



All files compiled with `+Owhole_program_mode` must also be compiled with `+O4`. If any of the files were compiled with `+O4`, but were not compiled with `+Owhole_program_mode`, the linker disables the assertion for all files in the program.

The default, `+Onowhole_program_mode`, disables the assertion noted above.

### **+tm *target***

**Optimization level:** `+O0`, `+O1`, `+O2`, `+O3`, `+O4`

**Default *target* value:** corresponds to the machine on which you invoke the compiler.

This option specifies the target machine architecture for which compilation is to be performed. Using this option causes the compiler to perform architecture-specific optimizations.

*target* takes one of the following values:

- `K8000` to specify K-Class servers using PA-8000 processors
- `V2000` to specify V2000 servers
- `V2200` to specify V2200 servers
- `V2250` to specify V2250 servers

This option is valid at all optimization levels. The default *target* value corresponds to the machine on which you invoke the compiler.

Using the `+tm target` option implies `+DA` and `+DS` settings as described in Table 18. `+DAarchitecture` causes the compiler to generate code for the architecture specified by *architecture*. `+DSmodel` causes the compiler to use the instruction scheduler tuned to *model*. See the `f90(1)` man page, `aCC(1)` page, or the `cc(1)` man page for more information describing the `+DA` and `+DS` options.

**Table 18**

**+tm *target* and +DA/+DS**

<b><i>target</i> value specified</b>	<b>+DA<i>architecture</i> implied</b>	<b>+DS<i>model</i> implied</b>
K8000	2.0	2.0
V2000	2.0	2.0
V2200	2.0	2.0
V2250	2.0	2.0

If you specify +DA or +DS on the compiler command line, your setting takes precedence over the setting implied by +tm *target*.

---

## C aliasing options

The optimizer makes a conservative assumption that a pointer can point to any object in the entire application. Command-line options to the C compiler are available to inform the optimizer of an application's pointer usage. Using this information, the optimizer can generate more efficient code, due to the elimination of some false assumptions.

You can direct pointer behavior to the optimizer by using the following options:

- `+O[no]ptrs_strongly_typed`
- `+O[no]ptrs_to_globals[=namelist]`
- `+O[no]global_ptr_unique[=namelist]`
- `+O[no]ptrs_ansi`

where

*namelist* is a comma-separated list of global variable names.

The following are type-inferred aliasing rules that apply when using these `+O` optimization options:

- Type-aliasing optimizations are based on the assumption that pointer dereferences obey their declared types.
- A C variable is considered address-exposed if and only if the address of that variable is assigned to another variable or passed to a function as an actual parameter. In general, address-exposed objects are collected into a separate group, based on their declared types. Global and static variables are considered address-exposed by default. Local variables and actual parameters are considered address-exposed only if their addresses have been computed using the address operator `&`.
- Dereferences of pointers to a certain type are assumed to only alias with the corresponding equivalent group. An equivalent group includes all the address-exposed objects of the same type. The dereferences of pointers are also assumed to alias with other pointer dereferences associated with the same group.

## Controlling optimization

### C aliasing options

For example, in the following line:

```
int *p, *q;
```

`*p` and `*q` are assumed to alias with any objects of type `int`. Also, `*p` and `*q` are assumed to alias with each other.

- Signed/unsigned type distinctions are ignored in grouping objects into an equivalent group. Likewise, `long` and `int` types are considered to map to the same equivalent group. However, the `volatile` type qualifier is considered significant in grouping objects into equivalent groups. For example, a pointer to `int` is not considered to alias with a `volatile int` object.
- If two type names reduce to the same type, they are considered synonymous.

In the following example, both types `type_old` and `type_new` reduce to the same type, `struct foo`.

```
typedef struct foo_st type_old;  
typedef type_old type_new;
```

- Each field of a structure type is placed in a separate equivalent group that is distinct from the equivalent group of the field's base type. The assumption here is that a pointer to `int` is not assigned the address of a structure field whose type is `int`. The actual type name of a structure type is not considered significant in constructing equivalent groups. For example, dereferences of a `struct foo` pointer and a `struct bar` pointer is assumed to alias with each other even if `struct foo` and `struct bar` have identical field declarations.
- All fields of a union type are placed in the same equivalent group, which is distinct from the equivalent group of any of the field's base types. This means that all dereferences of pointers to a particular union type are assumed to alias with each other, regardless of which union field is being accessed.
- Address-exposed array variables are grouped into the equivalent group of the array element type.
- Applying an explicit pointer typecast to an expression value causes any later use of the typecast expression value to be associated with the equivalent group of the typecast expression value.

For example, an `int` pointer typecast into a `float` pointer and then dereferenced is assumed to potentially access objects in the `float` equivalent group—and not the `int` equivalent group.

However, type-incompatible assignments to pointer variables do not alter the aliasing assumptions on subsequent references of such pointer variables.

In general, type-incompatible assignments can potentially invalidate some of the type-safe assumptions. Such constructs may elicit compiler warning messages.

---

## Optimization directives and pragmas

This section lists the directives, and pragmas available for use in optimization. Table 19 below describes the options and the optimization levels at which they are used. The pragmas are not supported by the aC++ compiler.

The `loop_parallel`, `parallel`, `prefer_parallel`, and `end_parallel` options are described in “Parallel programming techniques,” on page 175.

**Table 19**

**Directive-based optimization options**

Directives and Pragmas	Valid Optimization levels
<code>block_loop [(block_factor=<i>n</i>)]</code>	+03, +04
<code>dynsel[(trip_count=<i>n</i>)]</code>	+03, +04
<code>no_block_loop</code>	+03, +04
<code>no_distribute</code>	+03, +04
<code>no_dynsel</code>	+03, +04
<code>no_loop_dependence(<i>namelist</i>)</code>	+03, +04
<code>no_loop_transform</code>	+03, +04
<code>no_parallel</code>	+03, +04
<code>no_side_effects</code>	+03, +04
<code>no_unroll_and_jam</code>	+03, +04
<code>reduction(<i>namelist</i>)</code>	+03, +04
<code>scalar</code>	+03, +04
<code>sync_routine(<i>routinelist</i>)</code>	+03, +04
<code>unroll_and_jam[(unroll_factor=<i>n</i>)]</code>	+03, +04

## Rules for usage

The form of the optimization directives and pragmas is shown in Table 20.

**NOTE**

The HP aC++ compiler does not support the optimization pragmas described in this section.

**Table 20**

**Form of optimization directives and pragmas**

Language	Form
Fortran	C\$DIR <i>directive-list</i>
C	#pragma _CNX <i>directive-list</i>

where

*directive-list*

is a comma-separated list of one or more of the directives/pragmas described in this chapter.

- Directive names are presented here in lowercase, and they may be specified in either case in both languages. However, #pragma must always appear in lowercase in C.
- In the sections that follow, *namelist* represents a comma-separated 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 lowercase *n* or *m* is used to indicate an integer constant.
- Occurrences of *gate\_var* are for variables that have been or are being defined as gates. Any parameters that appear within square brackets ([ ]) are optional.

### **block\_loop[ (block\_factor=*n*) ]**

`block_loop[ (block_factor=n) ]` indicates a specific loop to block and, optionally, the block factor *n*. This block factor is used in the compiler's internal computation of loop nest-based data reuse; this is the number of times that the data reuse has resulted as a result of loop nesting. This figure must be an integer constant greater than or equal to 2. If no `block_factor` is specified, the compiler uses a heuristic to determine the `block_factor`. For more information on loop blocking, refer to "Optimization levels" on page 25.

### **dynsel[ (trip\_count=*n*) ]**

`dynsel[ (trip_count=n) ]` enables workload-based dynamic selection for the immediately following loop. *trip\_count* represents the `thread_trip_count` attribute, and *n* is an integer constant.

- When `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.
- For more information on dynamic selection, refer to the description of the optimization option "+O[no]dynsel" on page 120.

### **no\_block\_loop**

`no_block_loop` disables loop blocking on the immediately following loop. For more information on loop blocking, see the description of `block_loop[ (block_factor=n) ]` in this section, or refer to the description of the optimization option "+O[no]loop\_block" on page 127.

### **no\_distribute**

`no_distribute` disables loop distribution for the immediately following loop. For more information on loop distribution, refer to the description of the optimization option "+O[no]loop\_transform" on page 127.



### **no\_dynsel**

`no_dynsel` disables workload-based dynamic selection for the immediately following loop. For more information on dynamic selection, refer to the description of the optimization option “+O[no]dynsel” on page 120.

### **no\_loop\_dependence ( *namelist* )**

`no_loop_dependence ( namelist )` informs the compiler that the arrays in *namelist* do not have any dependences for iterations of the immediately following loop. Use `no_loop_dependence` for arrays only. Use `loop_private` to indicate dependence-free scalar variables.

This directive or pragma causes the compiler to ignore any dependences that it perceives to exist. This can enhance the compiler’s ability to optimize the loop, including parallelization.

For more information on loop dependence, refer to “Loop-carried dependences” on page 284.

### **no\_loop\_transform**

`no_loop_transform` prevents the compiler from performing reordering transformations on the following loop. The compiler does not distribute, fuse, block, interchange, unroll, unroll and jam, or parallelize a loop on which this directive appears. For more information on `no_loop_transform`, refer to the optimization option “+O[no]loop\_transform” on page 127.

### **no\_parallel**

`no_parallel` prevents the compiler from generating parallel code for the immediately following loop. For more information on `no_parallel`, refer to the optimization option “+O[no]parallel” on page 129.

### **no\_side\_effects(*funclist*)**

`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. The compiler may also be able to parallelize loops with calls when informed that the called routines do not have side effects.

### **unroll\_and\_jam[ (unroll\_factor=*n*) ]**

`unroll_and_jam[ (unroll_factor=n) ]` causes one or more noninnermost 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 noninnermost after any compiler-initiated interchanges. For more information on `unroll_and_jam`, refer to the description of “+O[no]loop\_unroll\_jam” on page 128.

---

# 8

# Optimization Report

The Optimization Report is produced by the HP Fortran 90, HP aC++, and HP C compilers. It is most useful at optimization levels +03 and +04. This chapter includes a discussion of the following topics:

- Optimization Report contents
- Loop Report

---

## Optimization Report contents

When you compile a program with the `+Oreport[=report_type]` optimization option at the `+O3` and `+O4` levels, the compiler generates an Optimization Report for each program unit. The `+Oreport[=report_type]` option determines the report's contents based on the value of *report\_type*, as shown in Table 21.

**Table 21** Optimization Report contents

<i>report_type</i> values	Report contents
all	Loop Report and Privatization Table
loop	Loop Report
private	Loop Report and Privatization Table
<i>report_type</i> not given (default)	Loop Report

The `+Onoreport` option does not accept any of the *report\_type* values. Sample Optimization Reports are provided throughout this chapter.

## Loop Report

The Loop Report lists the optimizations that are performed on loops and calls. If appropriate, the report gives 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.

Below is a sample optimization report.

Optimization Report					
Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
3	1	sub1	*Inlined call	(2-4)	
8	2	iloopi:1	Serial		Fused
11	3	jloopi:2	Serial		Fused
14	4	kloopi:3	Serial		Fused
			*Fused	(5)	(2 3 4) -> (5)
8	5	iloopi:1	PARALLEL		
Footnoted	User				
Var Name	Var Name				
iloopi:1	iloopindex				
jloopi:2	jloopindex				
kloopi:3	kloopindex				
Optimization for sub1					
Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
8	1	iloopi:1	Serial		Fused
11	2	jloopi:2	Serial		Fused
14	3	kloopi:3	Serial		Fused
			*Fused	(4)	(1 2 3) -> (4)
8	4	iloopi:1	PARALLEL		
Footnoted	User				
Var Name	Var Name				
iloopi:1	iloopindex				
jloopi:2	jloopindex				
kloopi:3	kloopindex				

A description of each column of the Loop Report is shown in Table 22.

**Table 22**      **Loop Report column definitions**

<b>Column</b>	<b>Description</b>
Line Num.	Specifies the source line of the beginning of the loop or of the loop from which it was derived. For cloned calls and inlined calls, the Line Num. column specifies the source line at which the call statement appears.
Id Num.	Specifies a unique ID number for every optimized loop and for every optimized call. 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 shown in the New Id Num.s column as described later. 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.
Var Name	Specifies the name of the iteration variable controlling the loop or the called procedure if the line represents a call. If the variable is compiler-generated, its name is listed as *VAR*. 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 Report and Analysis Table in the Optimization Report.
Reordering Transformation	Indicates which reordering transformations were performed. Reordering transformations are performed on loops, calls, 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 23 on page 155.
New Id Num.s	Specifies the ID number for loops or calls created by the compiler. These ID numbers are listed in the Id Num. column and is referenced in other parts of the report. However, the loops and calls they represent were not present in the original source code. In the case of loop fusion, the number in this column indicates the new loop created by merging all the fused loops. New ID numbers are also created for cloned calls, inlined calls, loop blocking, loop distribution, loop interchange, loop unroll and jam, dynamic selection, and test promotion.

Column	Description
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 24 on page 157.

The following values apply to the Reordering Transformation column described in Table 22 on page 154.

**Table 23 Reordering transformation values in the Loop Report**

Value	Description
Block	Loop blocking was performed. The new loop order is indicated under the Optimizing/Special Transformation column, as shown in Table 24.
Cloned call	A call to a subroutine was cloned.
Dist	Loop distribution was performed.
DynSel	Dynamic selection was performed. The numbers in the New Id Numms column correspond to the loops created. For parallel loops, these generally include a PARALLEL and a Serial version.
Fused	The loops were fused into another loop and no longer exist. The original loops and the new loop is indicated under the Optimizing/Special Transformation column, as shown in Table 24.
Inlined call	A call to a subroutine was inlined.
Interchange	Loop interchange was performed. The new loop order is indicated under the Optimizing/Special Transformation column, as shown in Table 24.
None	No reordering transformation was performed on the call.
PARALLEL	The loop runs in thread-parallel mode.
Peel	The first or last iteration of the loop was peeled in order to fuse the loop with an adjacent loop.
Promote	Test promotion was performed.

Optimization Report  
Loop Report

<b>Value</b>	<b>Description</b>
Serial	No reordering transformation was performed on the loop.
Unroll and Jam	The loop was unrolled and the nested loops were jammed (fused).
VECTOR	The loop was fully or partially replaced with more efficient calls to one or more vector routines.
*	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, blocking, fusion, interchange, call cloning, call inlining, peeling, promotion, unroll and jam).

The following values apply to the `Optimizing/special transformations` column described in Table 22 on page 154.



**Table 24**                    **Optimizing/special transformations values in the Loop Report**

Value	Explanation
Fused	The loop was fused into another loop and no longer exists.
Reduction	The compiler recognized a reduction in the loop.
Removed	The compiler removed the loop.
Unrolled	The loop was completely unrolled.
<i>(OrigOrder) -&gt; (InterchangedOrder)</i>	This information appears when <i>Interchange</i> is reported under <i>Reordering Transformation</i> . <i>OrigOrder</i> indicates the order of loops in the original nest. <i>InterchangedOrder</i> indicates the new order that occurs due to interchange. <i>OrigOrder</i> and <i>InterchangedOrder</i> consist of user iteration variables presented in outermost to innermost order.
<i>(OrigLoops)-&gt;(NewLoop)</i>	This information appears when <i>Fused</i> is reported under <i>Reordering Transformation</i> . <i>OrigLoops</i> indicates the original loops that were fused by the compiler to form the loop indicated by <i>NewLoop</i> . <i>OrigLoops</i> and <i>NewLoop</i> refer to loops based on the values from the <i>Id Num.</i> and <i>New Id Num.s</i> columns in the Loop Report.
<i>(OrigLoopNest)-&gt;(BlockedLoopNest)</i>	This information appears when <i>Block</i> is reported under <i>Reordering Transformation</i> . <i>OrigLoopNest</i> indicates the order of the original loop nest containing a loop that was blocked. <i>BlockedLoopNest</i> indicates the order of loops after blocking. <i>OrigLoopNest</i> and <i>BlockedLoopNest</i> refer to user iteration variables presented in outermost to innermost order.

## Supplemental tables

The tables described in this section may be included in the Optimization Report to provide information supplemental to the Loop Report.

### Analysis Table

If necessary, an Analysis Table is included in the Optimization Report to further elaborate on optimizations reported in the Loop Report.

A description of each column in the Analysis Table is shown in Table 25.

**Table 25**      **Analysis Table column definitions**

<b>Column</b>	<b>Description</b>
Line Num.	Specifies the source line of the beginning of the loop or call.
Id Num.	References the ID number assigned to the loop or call in the Loop Report.
Var Name	Specifies the name of the iteration variable controlling the loop, *VAR* (as discussed in the Var Name description in the section “Loop Report” on page 153).
Analysis	Indicates why a transformation or optimization was not performed, or additional information on what was done.

## 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 Report is automatically provided with it.

A description of each column in the Privatization Table is shown in Table 26.

**Table 26**

**Privatization Table column definitions**

Column	Definitions
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.
Var Name	Specifies the name of the iteration variable controlling the loop. *VAR* may also appear in this column, as discussed in the Var Name description in the section “Loop Report” on page 153.
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 variable privatizations performed.

## Variable Name Footnote Table

Variable names that are too long to fit in the `Var Name` 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.

A description of each column in the Variable Name Footnote Table is shown in Table 27.

**Table 27** Variable Name Footnote Table column definitions

Column	Definition
Footnoted Var Name	Specifies the truncated variable name and its footnote number.
User Var Name	Specifies the full name of the variable as identified in the source code.

### Example

#### Optimization Report

The following Fortran program is the basis for the Optimization Report shown in this example. Line numbers are provided for ease of reference.

```
1 PROGRAM EXAMPLE99
2 REAL A(100), B(100), C(100)
3 CALL SUB1(A,B,C)
4 END
5
6 SUBROUTINE SUB1(A,B,C)
7 REAL A(100), B(100), C(100)
8 DO ILOOPINDEX=1,100
9   A(ILOOPINDEX) = ILOOPINDEX
10 ENDDO
11 DO JLOOPINDEX=1,100
12   B(JLOOPINDEX) = A(JLOOPINDEX)**2
13 ENDDO
14 DO KLOOPINDEX=1, 100
15   C(KLOOPINDEX) = A(KLOOPINDEX) + B(KLOOPINDEX)
16 ENDDO
17 PRINT *, A(1), B(50), C(100)
18 END
```

The following Optimization Report is generated by compiling the program `EXAMPLE99` with the command-line options `+O3 +Oparallel +Oreport=all +Oinline=sub1`:

```
% f90 +O3 +Oparallel +Oreport=all +Oinline=sub1 EXAMPLE99.f
```

Optimization for EXAMPLE99

Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
3	1	sub1	*Inlined call	(2-4)	
8	2	iloopi:1	Serial		Fused
11	3	jloopi:2	Serial		Fused
14	4	kloopi:3	Serial		Fused
8	5	iloopi:1	PARALLEL	(5)	(2 3 4) -> (5)
Footnoted Var Name	User Var Name				
iloopi:1	iloopindex				
jloopi:2	jloopindex				
kloopi:3	kloopindex				

Optimization for sub1

Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
8	1	iloopi:1	Serial		Fused
11	2	jloopi:2	Serial		Fused
14	3	kloopi:3	Serial		Fused
8	4	iloopi:1	PARALLEL	(4)	(1 2 3) -> (4)
Footnoted Var Name	User Var Name				
iloopi:1	iloopindex				
jloopi:2	jloopindex				
kloopi:3	kloopindex				

The Optimization Report for EXAMPLE99 provides the following information:

- **Call to sub1 is inlined**  
 The first line of the Loop Report shows that the call to sub1 was inlined, as shown below:  

```
3      1  sub1      *Inlined call      (2-4)
```
- **Three new loops produced**  
 The inlining produced three new loops in EXAMPLE99: Loop #2, Loop #3, and Loop #4. Internally, the EXAMPLE99 module that originally looked like:

## Optimization Report

### Loop Report

```
1 PROGRAM EXAMPLE99
2 REAL A(100), B(100), C(100)
3 CALL SUB1(A,B,C)
4 END
```

now looks like this:

```
PROGRAM EXAMPLE99
REAL A(100), B(100), C(100)
DO ILOOPINDEX=1,100 !Loop #2
  A(ILOOPINDEX) = ILOOPINDEX
ENDDO
DO JLOOPINDEX=1,100 !Loop #3
  B(JLOOPINDEX) = A(JLOOPINDEX)**2
ENDDO
DO KLOOPINDEX=1, 100 !Loop #4
  C(KLOOPINDEX) = A(KLOOPINDEX) + B(KLOOPINDEX)
ENDDO
PRINT *, A(1), B(50), C(100)
END
```

- **New loops are fused**

These lines indicate that the new loops have been fused. The following line indicates that the three loops were fused into one new loop, Loop #5.

```
8      2  iloopi:1  Serial      Fused
11     3  jloopi:2  Serial      Fused
14     4  kloopi:3  Serial      Fused
                                *Fused      (5)      (2 3 4) (5)
```

After fusing, the code internally appears as the following:

```
PROGRAM EXAMPLE99
REAL A(100), B(100), C(100)
DO ILOOPINDEX=1,100 !Loop #5
  A(ILOOPINDEX) = ILOOPINDEX
  B(ILOOPINDEX) = A(ILOOPINDEX)**2
  C(ILOOPINDEX) = A(ILOOPINDEX) + B(ILOOPINDEX)
ENDDO
PRINT *, A(1), B(50), C(100)
END
```

- **New loop is parallelized**

In the following Loop Report line:

```
8          5  iloopi:1  PARALLEL
```

Loop #5 uses `iloopi:1` as the iteration variable, referencing the Variable Name Footnote Table; `iloopi:1` corresponds to `iloopindex`. The same line in the report also indicates that the newly-created Loop #5 was parallelized.

- **Variable Name Footnote Table lists iteration variables**

According to the Variable Name Footnote Table (duplicated below), the original variable `iloopindex` is abbreviated by the compiler as `iloopi:1` so that it fits into the Var Name columns of other reports.

`jloopindex` and `kloopindex` are abbreviated as `jloopi:2` and `kloopi:3`, respectively. These names are used throughout the report to refer to these iteration variables.

Footnoted Var Name	User Var Name
-----	
<code>iloopi:1</code>	<code>iloopindex</code>
<code>jloopi:2</code>	<code>jloopindex</code>
<code>kloopi:3</code>	<code>kloopindex</code>

## Example

### Optimization Report

The following Fortran code provides an example of other transformations the compiler performs. Line numbers are provided for ease of reference.

```
1  PROGRAM EXAMPLE100
2
3  INTEGER IA1(100), IA2(100), IA3(100)
4  INTEGER I1, I2
5
6  DO I = 1, 100
7    IA1(I) = I
8    IA2(I) = I * 2
9    IA3(I) = I * 3
10 ENDDO
11
12 I1 = 0
13 I2 = 100
14 CALL SUB1 (IA1, IA2, IA3, I1, I2)
15 END
16
17 SUBROUTINE SUB1(A, B, C, S, N)
18 INTEGER A(N), B(N), C(N), S, I, J
19 DO J = 1, N
20 DO I = 1, N
21 IF (I .EQ. 1) THEN
```

Optimization Report  
Loop Report

```

22         S = S + A(I)
23     ELSE IF (I .EQ. N) THEN
24         S = S + B(I)
25     ELSE
26         S = S + C(I)
27     ENDIF
28     ENDDO
29 ENDDO
30 END

```

The following Optimization Report is generated by compiling the program EXAMPLE100 for parallelization:

```
% f90 +O3 +Oparallel +Oreport=all example100.f
```

Optimization for SUB1

Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
19	1	j	*Interchange	(2)	(j i) -> (i j)
20	2	i	*DynSel	(3-4)	
20	3	i	PARALLEL		Reduction
19	5	j	*Promote	(6-7)	
19	6	j	Serial		
19	7	j	Serial		
20	4	i	Serial		
19	8	j	*Promote	(9-10)	
19	9	j	Serial		
19	10	j	*Promote	(11-12)	
19	11	j	Serial		
19	12	j	Serial		

Line Num.	Id Num.	Var Name	Analysis
19	5	j	Test on line 21 promoted out of loop
19	8	j	Test on line 21 promoted out of loop
19	10	j	Test on line 23 promoted out of loop

The report is continued on the next page.

Optimization for clone 1 of SUB1 (6\_e70\_cl\_sub1)

Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
19	1	j	*Interchange	(2)	(j i) -> (i j)
20	2	i	PARALLEL		Reduction
19	3	j	*Promote	(4-5)	
19	4	j	Serial		
19	5	j	*Promote	(6-7)	
19	6	j	Serial		
19	7	j	Serial		

Line	Id	Var	Analysis
------	----	-----	----------



Num.	Num.	Name			
19	3	j	Test on line 21 promoted out of loop		
19	5	j	Test on line 23 promoted out of loop		
Optimization for example100					
Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
6	1	i	Serial		
14	2	sub1	*Cloned call	(3)	
14	3	sub1	None		
Line Num.	Id Num.	Var Name	Analysis		
14	2	sub1	Call target changed to clone 1 of SUB1 (6_e70_cl_sub1)		

The Optimization Report for EXAMPLE100 shows Optimization Reports for the subroutine and its clone, followed by the optimizations to the subroutine. It includes the following information:

- **Original subroutine contents**

Originally, the subroutine appeared as shown below:

```

17  SUBROUTINE SUB1(A, B, C, S, N)
18  INTEGER A(N), B(N), C(N), S, I, J
19  DO J = 1, N
20  DO I = 1, N
21  IF (I .EQ. 1) THEN
22  S = S + A(I)
23  ELSE IF (I .EQ. N) THEN
24  S = S + B(I)
25  ELSE
26  S = S + C(I)
27  ENDIF
28  ENDDO
29  ENDDO
30  END

```

- **Loop interchange performed first**

The compiler first performs loop interchange (listed as Interchange in the report) to maximize cache performance:

```

19  1 j *Interchange (2) (j i) -> (i j)

```

Optimization Report  
Loop Report

- **The subroutine then becomes the following**

```
17  SUBROUTINE SUB1(A, B, C, S, N)
18  INTEGER A(N), B(N), C(N), S, I, J
19      DO I = 1, N                                ! Loop #2
20          DO J = 1, N                                ! Loop #1
21              IF (I .EQ. 1) THEN
22                  S = S + A(I)
23              ELSE IF (I .EQ. N) THEN
24                  S = S + B(I)
25              ELSE
26                  S = S + C(I)
27              ENDIF
28          ENDDO
29      ENDDO
30  END
```

- **The program is optimized for parallelization**

The compiler would like to parallelize the outermost loop in the nest, which is now the `I` loop. However because the value of `N` is not known, the compiler does not know how many times the `I` loop needs to be executed. To ensure that the loop is executed as efficiently as possible at runtime, the compiler replaces the `I` loop nest with two new copies of the `I` loop nest, one to be run in parallel, the other to be run serially.

- **Dynamic selection is executed**

An `IF` is then inserted to select the more efficient version of the loop to execute at runtime. This method of making one copy for parallel execution and one copy for serial execution is known as dynamic selection, which is enabled by default when `+O3 +Oparallel` is specified (see “Dynamic selection” on page 102 for more information). This optimization is reported in the Loop Report in the line:

```
20          2 i          *DynSel          (3-4)
```

- **Loop#2 creates two loops**

According to the report, Loop #2 was used to create the new loops, Loop #3 and Loop #4. Internally, the code now is represented as follows:

```
SUBROUTINE SUB1(A, B, C, S, N)
INTEGER A(N), B(N), C(N), S, I, J
IF (N .GT. some_threshold) THEN
```

```

DO (parallel) I = 1, N                ! Loop #3
DO J = 1, N                          ! Loop #5
  IF (I .EQ. 1) THEN
    S = S + A(I)
  ELSE IF (I .EQ. N) THEN
    S = S + B(I)
  ELSE
    S = S + C(I)
  ENDIF
ENDDO
ENDDO
ELSE
DO I = 1, N                          ! Loop #4
DO J = 1, N                          ! Loop #8
  IF (I .EQ. 1) THEN
    S = S + A(I)
  ELSE IF (I .EQ. N) THEN
    S = S + B(I)
  ELSE
    S = S + C(I)
  ENDIF
ENDDO
ENDDO
ENDIF
END

```

- **Loop#3 contains reductions**

Loop #3 (which was parallelized) also contained one or more reductions. The Reordering Transformation column indicates that the IF statements were promoted out of Loop #5, Loop #8, and Loop #10.

- **Analysis Table lists new loops**

The line numbers of the promoted IF statements are listed. The first test in Loop #5 was promoted, creating two new loops, Loop #6 and Loop #7. Similarly, Loop #8 has a test promoted, creating Loop #9 and Loop #10. The test remaining in Loop #10 is then promoted, thereby creating two additional loops. A promoted test is an IF statement that is hoisted out of a loop. See the section “Test promotion” on page 90 for more information. The Analysis Table contents are shown below:

19	5	j	Test on line 21 promoted out of loop
19	8	j	Test on line 21 promoted out of loop
19	10	j	Test on line 23 promoted out of loop

Optimization Report  
Loop Report

- **DO loop is not reordered**

The following DO loop does not undergo any reordering transformation:

```
6      DO I = 1, 100
7          IA1(I) = I
8          IA2(I) = I * 2
9          IA3(I) = I * 3
10     ENDDO
```

This fact is reported by the line

```
6          1  i          Serial
```

- **sub1 is cloned**

The call to the subroutine sub1 is cloned. As indicated by the asterisk (\*), the compiler produced a new call. The new call is given the ID (3) listed in the New Id Nums column. The new call is then listed, with None indicating that no reordering transformation was performed on the call to the new subroutine.

```
14          2  sub1      *Cloned call      (3)
14          3  sub1      None
```

- **Cloned call is transformed**

The call to the subroutine is then appended to the Loop Report to elaborate on the Cloned call transformation. This line shows that the clone was called in place of the original subroutine.

```
14  2  sub1  Call target changed to clone 1 of SUB1 (6_e70_cl_sub1)
```

## Example

### Optimization Report

The following Fortran code shows loop blocking, loop peeling, loop distribution, and loop unroll and jam. Line numbers are listed for ease of reference.

```
1      PROGRAM EXAMPLE200
2
3      REAL*8 A(1000,1000), B(1000,1000), C(1000)
4      REAL*8 D(1000), E(1000)
5      INTEGER M, N
6
7      N = 1000
8      M = 1000
9
10     DO I = 1, N
11         C(I) = 0
12         DO J = 1, M
13             A(I,J) = A(I,J) + B(I,J) * C(I)
14         ENDDO
15     ENDDO
16
17     DO I = 1, N-1
18         D(I) = I
19     ENDDO
20
21     DO J = 1, N
22         E(J) = D(J) + 1
23     ENDDO
24
25     PRINT *, A(103,103), B(517, 517), D(11), E(29)
26
27     END
```

The following Optimization Report is generated by compiling program EXAMPLE200 as follows:

```
% f90 +O3 +Oreport +Oloop_block example200.f
```

## Optimization Report Loop Report

Optimization for example3

Line Num.	Id Num.	Var Name	Reordering Transformation	New Id Nums	Optimizing / Special Transformation
10	1	i:1	*Dist	(2-3)	
10	2	i:1	Serial		
10	3	i:1	*Interchange	(4)	(i:1 j:1) -> (j:1 i:1)
12	4	j:1	*Block	(5)	(j:1 i:1) -> (i:1 j:1 i:1)
10	5	i:1	*Promote	(6-7)	
10	6	i:1	Serial		Removed
10	7	i:1	Serial		
12	8	j:1	*Unroll And Jam	(9)	
12	9	j:1	*Promote	(10-11)	
12	10	j:1	Serial		Removed
12	11	j:1	Serial		
10	12	i:1	Serial		
17	13	i:2	Serial		Fused
21	14	j:2	*Peel	(15)	
21	15	j:2	Serial		Fused
			*Fused	(16)	(13 15) -> (16)
17	16	i:2	Serial		

Line Num.	Id Num.	Var Name	Analysis
10	5	i:1	Loop blocked by 56 iterations
10	5	i:1	Test on line 12 promoted out of loop
10	6	i:1	Loop blocked by 56 iterations
10	7	i:1	Loop blocked by 56 iterations
12	8	j:1	Loop unrolled by 8 iterations and jammed into the innermost loop
12	9	j:1	Test on line 10 promoted out of loop
21	14	j:2	Peeled last iteration of loop

The Optimization Report for EXAMPLE200 provides the following results:

```
10      1  i:1      *Dist      (2-3)
```

- **Several occurrences of variables noted**

In this report, the Var Name column has entries such as i:1, j:1, i:2, and j:2. This type of entry appears when a variable is used more than once. In EXAMPLE200, I is used as an iteration variable twice. Consequently, i:1 refers to the first occurrence, and i:2 refers to the second occurrence.

- **Loop #1 creates new loops**

The first line of the report shows that Loop #1, shown on line 10, is distributed to create Loop #2 and Loop #3:

Initially, Loop #1 appears as shown.

```
DO I = 1, N                                ! Loop #1
  C(I) = 0
  DO J = 1, M
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

It is then distributed as follows:

```
DO I = 1, N                                ! Loop #2
  C(I) = 0
ENDDO

DO I = 1, N                                ! Loop #3
  DO J = 1, M
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

- **Loop #3 is interchanged to create Loop#4**

The third line indicates this:

```
10      3  i:1      *Interchange      (4)      (i:1 j:1) ->
(j:1 i:1)
```

Now, the loop looks like the following code:

```
DO J = 1, M                                ! Loop #4
  DO I = 1, N
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

- **Nested loop is blocked**

The next line of the Optimization Report indicates that the nest rooted at Loop #4 is blocked:

```
12      4  j:1      *Block              (5)      (j:1 i:1) ->
(i:1 j:1 i:1)
```

The blocked nest internally appears as follows:

```
DO IOU = 1, N, 56                          ! Loop #5
  DO J = 1, M
    DO I = IOU, IOU + 55
      A(I,J) = A(I,J) + B(I,J) * C(I)
    ENDDO
  ENDDO
ENDDO
```

Optimization Report  
Loop Report

- **Loop #5 noted as blocked**  
The loop with iteration variable `i:1` is the loop that was actually blocked. The report shows `*Block on Loop #4` (the `j:1` loop) because the entire nest rooted at Loop #4 is replaced by the blocked nest.
- **IOUT variable facilitates loop blocking**  
The `IOUT` variable is introduced to facilitate the loop blocking. The compiler uses a step value of 56 for the `IOUT` loop as reported in the Analysis Table:

```
10      5  i:1      Loop blocked by 56 iterations
```

- **Test promotion creates new loops**  
The next three lines of the report show that a test was promoted out of Loop #5, creating Loop #6 (which is removed) and Loop #7 (which is run serially). This test—which does not appear in the source code—is an implicit test that the compiler inserts in the code to ensure that the loop iterates at least once.

```
10      5  i:1      *Promote      (6-7)
10      6  i:1      Serial          Removed
10      7  i:1      Serial
```

This test is referenced again in the following line from the Analysis Table:

```
10      5  i:1      Test on line 12 promoted out of loop
```

- **Unroll and jam creates new loop**  
The report indicates that the `J` is unrolled and jammed, creating Loop #9:

```
12      8  j:1      *Unroll And Jam   (9)
```

- **J loop unrolled by 8 iterations**  
This line also indicates that the `J` loop is unrolled by 8 iterations and fused:

```
12      8  j:1      Loop unrolled by 8 iterations and jammed
                  into the innermost loop
```



The unrolled and jammed loop results in the following code:

```

DO IOUT = 1, N, 56                                ! Loop #5
DO J = 1, M, 8                                    ! Loop #8
DO I = IOUT, IOUT + 55                            ! Loop #9
  A(I,J) = A(I,J) + B(I,J) * C(I)
  A(I,J+1) = A(I,J+1) + B(I,J+1) * C(I)
  A(I,J+2) = A(I,J+2) + B(I,J+2) * C(I)
  A(I,J+3) = A(I,J+3) + B(I,J+3) * C(I)
  A(I,J+4) = A(I,J+4) + B(I,J+4) * C(I)
  A(I,J+5) = A(I,J+5) + B(I,J+5) * C(I)
  A(I,J+6) = A(I,J+6) + B(I,J+6) * C(I)
  A(I,J+7) = A(I,J+7) + B(I,J+7) * C(I)
ENDDO
ENDDO
ENDDO

```

- **Test promotion in Loop #9 creates new loops**

The Optimization Report indicates that the compiler-inserted test in Loop #9 is promoted out the loop, creating Loop #10 and Loop #11.

```

12    9    j:1  *Promote    (10-11)
12   10    j:1   Serial          Removed
12   11    j:1   Serial

```

- **Loops are fused**

According to the report, the last two loops in the program are fused (once an iteration is peeled off the second loop), then the new loop is run serially.

```

17   13  i:2   Serial          Fused
21   14  j:2  *Peel          (15)
21   15  j:2   Serial          Fused
                                *Fused    (16)  (13 15) -> (16)
17   16  i:2   Serial

```

That information is combined with the following line from the Analysis Table:

```

21   14  j:2          Peeled last iteration of loop

```

- **Loop peeling creates loop, enables fusion**

Initially, Loop #14 has an iteration peeled to create Loop #15, as shown below. The loop peeling is performed to enable loop fusion.

```

DO I = 1, N-1                                    ! Loop #13
  D(I) = I
ENDDO

DO J = 1, N-1                                    ! Loop #15
  E(J) = D(J) + 1
ENDDO

```

Optimization Report  
Loop Report

- **Loops are fused to create new loop**

Loop #13 and Loop #15 are then fused to produce Loop #16:

```
DO I = 1, N-1                                ! Loop #16
  D(I) = I
  E(I) = D(I) + 1
ENDDO
```

---

# 9

## Parallel programming techniques

The HP compiler set provides programming techniques that allow you to increase code efficiency while achieving three-tier parallelism. This chapter describes the following programming techniques and requirements for implementing low-overhead parallel programs:

- Parallelizing directives and pragmas
- Parallelizing loops
- Parallelizing tasks
- Parallelizing regions
- Reentrant compilation
- Setting thread default stack size
- Collecting parallel information

### NOTE

The HP aC++ compiler does not support the pragmas described in this chapter.

---

## Parallelizing directives and pragmas

This section summarizes the directives and pragmas used to achieve parallelization in the HP compilers. The directives and pragmas are listed in the order of how they would typically be used within a given program.

**Table 28** Parallel directives and pragmas

Pragma / Directive	Description	Level of parallelism
<code>prefer_parallel</code> [ ( <i>attribute_list</i> ) ]	Requests parallelization of the immediately following loop, accepting attribute combinations for thread-parallelism, strip-length adjustment, and maximum number of threads. The compiler handles data privatization and does not parallelize the loop if it is not safe to do so.	Loop
<code>loop_parallel</code> [ ( <i>attribute_list</i> ) ]	Forces parallelization of the immediately following loop. Accepts attributes for thread-parallelism, strip-length adjustment, maximum number of threads, and ordered execution. Requires you to manually privatize loop data and synchronize data dependences.	Loop
<code>parallel</code> [ ( <i>attribute_list</i> ) ]	Allow you to parallelize a single code region to run on multiple threads. Unlike the tasking directives, which run discrete sections of code in parallel, <code>parallel</code> and <code>end_parallel</code> run multiple copies of a single section. Accepts attribute combinations for thread-parallelism and maximum number of threads.  Within a <code>parallel</code> region, loop directives ( <code>prefer_parallel</code> , <code>loop_parallel</code> ) and tasking directives ( <code>begin_tasks</code> ) may appear with the <code>dist</code> attribute.	Region
<code>end_parallel</code>	Signifies the end of a parallel region (see <code>parallel</code> ).	Region

<b>Pragma / Directive</b>	<b>Description</b>	<b>Level of parallelism</b>
<code>begin_tasks</code> ( <i>attribute_list</i> )	Defines the beginning of a series of tasks, allowing you to parallelize consecutive blocks of code. Accepts attribute combinations for thread-parallelism, ordered execution, maximum number of threads, and others.	Task
<code>next_task</code>	Starts a block of code following a <code>begin_tasks</code> block that will be executed as a parallel task.	Task
<code>end_tasks</code>	Terminates parallel tasks started by <code>begin_tasks</code> and <code>next_task</code> .	Task
<code>ordered_section</code> ( <i>gate</i> )	Allows you to isolate dependences within a loop so that code contained within the ordered section executes in iteration order. Only useful when used with <code>loop_parallel(ordered)</code> .	Loop
<code>critical_section</code> [( <i>gate</i> )]	Allows you to isolate nonordered manipulations of a shared variable within a loop. Only one parallel thread can execute the code contained in the critical section at a time, eliminating possible contention.	Loop
<code>end_critical_section</code>	Identifies the end of a critical section (see <code>critical_section</code> ).	Loop
<code>reduction</code>	Forces reduction analysis on a loop being manipulated by the <code>loop_parallel</code> directive. See “Reductions” on page 108.	Loop
<code>sync_routine</code>	Must be used to identify synchronization functions that you call indirectly call in your own routines. See “ <code>sync_routine</code> ” on page 242.	Loop or Task

## Parallelizing loops

The HP compilers automatically exploit loop parallelism in dependence-free loops. The `prefer_parallel`, `loop_parallel`, and `parallel` directives and pragmas allow you to increase parallelization opportunities and to manually control many aspects of parallelization using simple manual loop parallelization.

The `prefer_parallel` and `loop_parallel` directives and pragmas, apply to the immediately following loop. Data privatization is necessary when using `loop_parallel`; this is achieved by using the `loop_private` directive, discussed in “Data privatization,” on page 207. Manual data privatization using memory classes is discussed in “Memory classes,” on page 223 and “Parallel synchronization,” on page 233.

The parallel directives and pragmas should only be used on Fortran `DO` and C `for` loops that have iteration counts that are determined prior to loop invocation at runtime.

### `prefer_parallel`

The `prefer_parallel` directive and pragma causes the compiler to automatically parallelize the immediately following loop if it is free of dependences and other parallelization inhibitors. The compiler automatically privatizes any loop variables that must be privatized. `prefer_parallel` requires less manual intervention. However, it is less powerful than the `loop_parallel` directive and pragma.

See “`prefer_parallel`, `loop_parallel` attributes” on page 181 for a description of attributes for this directive.

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

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

This code indicates that `PREFER_PARALLEL` causes the compiler to choose the innermost loop for parallelization, provided it is free of dependences. `PREFER_PARALLEL` does not inhibit loop interchange.

The `ordered` attribute in a `prefer_parallel` directive is only useful if the loop contains synchronized dependences. The `ordered` attribute is most useful in the `loop_parallel` directive, described in the next section.

## `loop_parallel`

The `loop_parallel` directive forces parallelization of the immediately following loop. The compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must synchronize any dependences manually and manually privatize loop data as necessary. `loop_parallel` defaults to thread parallelization.

See “`prefer_parallel`, `loop_parallel` attributes” on page 181 for a description of attributes for this directive.

`loop_parallel(ordered)` is useful for manually parallelizing loops that contain ordered dependences. This is described in “Parallel synchronization,” on page 233.

## Parallelizing loops with calls

`loop_parallel` is useful for manually parallelizing loops containing procedure calls.

This is shown in the following Fortran code:

```
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. To verify that the `FUNC` has no side effects, review the following conditions. A function does not have side effects if:

- It does not modify its arguments.
- It does not modify the same memory location from one call to the next.
- It performs no I/O.

Parallel programming techniques  
Parallelizing loops

- It does not call any procedures that have side effects. If `FUNC` does have side effects or is not reentrant, this loop may yield wrong answers.

If you are sure that

`FUNC`

has no side effects and is compiled for reentrancy (the default), this loop can be safely parallelized.

NOTE

In some cases, global register allocation can interfere with the routine being called. Refer to the “Global register allocation (GRA)” on page 43 for more information.

### Unparallelizable loops

The compiler does not parallelize any loop that does not have a number of iterations that can be determined prior to loop invocation at execution time, even when `loop_parallel` is specified.

This is shown in the following Fortran code:

```
C$DIR LOOP_PARALLEL
DO WHILE(A(I) .GT. 0)!WILL NOT PARALLELIZE
  .
  .
  A(I) = ...
  .
  .
ENDDO
```

In general, there is no way the compiler can determine the loop’s iteration count prior to loop invocation here, so the loop cannot be parallelized.



### **prefer\_parallel, loop\_parallel attributes**

The `prefer_parallel` and `loop_parallel` directives and pragmas maintain the same attributes. The forms of these directives and pragmas are shown in Table 29.

**Table 29**      **Forms of `prefer_parallel` and `loop_parallel` directives and pragmas**

Language	Form
Fortran	<code>C\$DIR PREFER_PARALLEL[ ( <i>attribute-list</i> ) ]</code> <code>C\$DIR LOOP_PARALLEL[ ( <i>attribute-list</i> ) ]</code>
C	<code>#pragma _CNX prefer_parallel[ ( <i>attribute-list</i> ) ]</code> <code>#pragma _CNX loop_parallel(ivar = <i>indvar</i>[ , <i>attribute-list</i> ])</code>

where

`ivar = indvar`

specifies that the primary loop induction variable is *indvar*. `ivar = indvar` is optional in Fortran, but required in C. Use only with `loop_parallel`.

*attribute-list*

can contain one of the case-insensitive attributes noted in Table 30.

**NOTE**

The values of *n* and *m* must be compile-time constants for the loop parallelization attributes in which they appear.

**Table 30**      **Attributes for `loop_parallel`, `prefer_parallel`**

Attribute	Description
dist	<p>Causes the compiler to distribute the iterations of a loop across active threads instead of spawning new threads. This significantly reduces parallelization overhead.</p> <p>Must be used with <code>prefer_parallel</code> or <code>loop_parallel</code> inside a <code>parallel/end_parallel</code> region.</p> <p>Can be used with any <code>prefer_parallel</code> or <code>loop_parallel</code> attribute, except <code>threads</code>.</p>
ordered	<p>Causes the iterations of the loop to be initiated in iteration order across the processors. This is useful only in loops with manually-synchronized dependences, constructed using <code>loop_parallel</code>.</p> <p>To achieve ordered parallelism, dependences must be synchronized within ordered sections, constructed using the <code>ordered_section</code> and <code>end_ordered_section</code> directives.</p>
<code>max_threads = m</code>	<p>Restricts execution of the specified loop to no more than <i>m</i> threads if specified alone. <i>m</i> must be an integer constant.</p> <p><code>max_threads = m</code> is useful when you know the maximum number of threads your loop runs on efficiently.</p> <p>If specified with the <code>chunk_size = n</code> attribute, the chunks are parallelized across no more than <i>m</i> threads.</p>

Attribute	Description
<code>chunk_size = n</code>	<p>Divides the loop into chunks of <math>n</math> or fewer iterations by which to strip mine the loop for parallelization. <math>n</math> must be an integer constant.</p> <p>If <code>chunk_size = n</code> is present alone, <math>n</math> or fewer loop iterations are distributed round-robin to each available thread until there are no remaining iterations. This is shown in Table 32 and Table 33 on page 186.</p> <p>If the number of threads does not evenly divide the number of iterations, some threads perform one less chunk than others.</p>
<code>dist, ordered</code>	Causes ordered invocation of each iteration across existing threads.
<code>dist, max_threads = m</code>	Causes thread-parallelism on no more than $m$ existing threads.
<code>ordered, max_threads = m</code>	Causes ordered parallelism on no more than $m$ threads.
<code>dist, chunk_size = n</code>	Causes thread-parallelism by chunks.
<code>dist, ordered, max_threads = m</code>	Causes ordered thread-parallelism on no more than $m$ existing threads.
<code>chunk_size = n, max_threads = m</code>	Causes chunk parallelism on no more than $m$ threads.
<code>dist, chunk_size = n, max_threads = m</code>	Causes thread-parallelism by chunks on no more than $m$ existing threads.

Any loop under the influence of `loop_parallel(dist)` or `prefer_parallel(dist)` appears in the Optimization Report as `serial`. This is because it is already inside a parallel region. You can generate an Optimization Report by specifying the `+Oreport` option. For more information, see “Optimization Report,” on page 151.

## Combining the attributes

Table 30 shown above describes the acceptable combinations of `loop_parallel` and `prefer_parallel` attributes. In such combinations, the attributes are 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 issues a warning and ignores the pragma. `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 it is not provided, the compiler picks the primary induction variable for the loop. `ivar` is required for DO, WHILE and customized loops in Fortran.

### NOTE

`prefer_parallel` does not require `ivar`. The compiler issues an error if it encounters this combination.

## Comparing `prefer_parallel`, `loop_parallel`

The `prefer_parallel` and `loop_parallel` directives and pragmas are used to parallelize loops. Table 31 provides an overview of the differences between the two pragmas/directives. See the sections “`prefer_parallel`” on page 178 and “`loop_parallel`” on page 179 for more information.

**Table 31** Comparison of `loop_parallel` and `prefer_parallel`

	<code>prefer_parallel</code>	<code>loop_parallel</code>
<b>Description</b>	Requests compiler to perform parallelization analysis on the following loop then parallelize the loop if it is safe to do so. When used with the <code>+Oautopar</code> option (the default), it overrides the compiler heuristic for picking which loop in a loop nest to parallelize. When used with <code>+Onoautopar</code> , the compiler only performs directive-specified parallelization. No heuristic is used to pick the loop in a nest to parallelize. In such cases, <code>prefer_parallel</code> requests loop parallelization.	Forces the compiler to parallelize the following loop—assuming the iteration count can be determined prior to loop invocation.
<b>Advantages</b>	Compiler automatically performs parallelization analysis and variable privatization.	Allows you to parallelize loops that the compiler is not able to automatically parallelize because it cannot determine dependences or side effects.
<b>Disadvantages</b>	Loop may or may not execute in parallel.	Requires you to: —Check for and synchronize any data dependences —Perform variable privatization

## Stride-based parallelism

Stride-based parallelism differs from the default strip-based parallelism described in that:

- Strip-based parallelism divides the loop's iterations into a number of contiguous chunks equal to the number of available threads, and each thread computes one chunk.
- Stride-based parallelism, set by the `chunk_size=n` attribute, allows each thread to do several noncontiguous chunks.

Specifying `chunk_size = ((number of iterations - 1) / number of threads) + 1` is similar to default strip mining for parallelization.

Using `chunk_size = 1` distributes individual iterations cyclically across the processors. For example, if a loop has 1000 iterations to be distributed among 4 processors, specifying `chunk_size=1` causes the distribution shown in Table 32.

**Table 32** Iteration distribution using `chunk_size = 1`

	<b>CPU0</b>	<b>CPU1</b>	<b>CPU2</b>	<b>CPU3</b>
<b>Iterations</b>	1	2	3	4
	5	6	7	...

For `chunk_size=n`, with  $n > 1$ , the distribution is round-robin. However, it is not the same as specifying the `ordered` attribute. For example, using the same loop as above, specifying `chunk_size=5` produces the distribution shown in Table 33.

**Table 33** Iteration distribution using `chunk_size = 5`

	<b>CPU0</b>	<b>CPU1</b>	<b>CPU2</b>	<b>CPU3</b>
<b>Iterations</b>	1, 2, 3, 4, 5	6, 7, 8, 9, 10	11, 12, 13, 14, 15	16, 17, 18, 19, 20
	21, 22, 23, 24, 25	26, 27, 28, 29, 30	31, 32, 33, 34, 35,	...

For more information and examples on using the `chunk_size = n` attribute, see "Troubleshooting," on page 265.

**Example**

`prefer_parallel, loop_parallel`

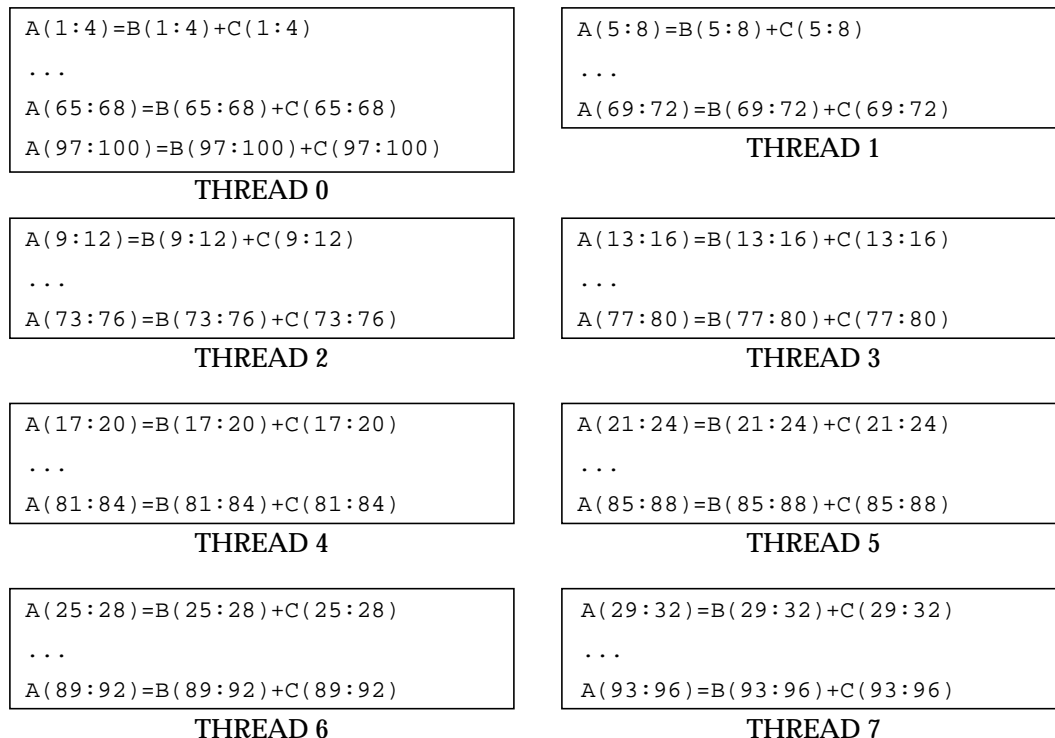
The following Fortran example uses the `PREFER_PARALLEL` directive, but applies to `LOOP_PARALLEL` as well:

```
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 four iterations to each available thread. Figure 16 uses Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming eight available threads.

Figure 16 shows that the 100 iterations of `I` are parcelled out in chunks of four iterations to each of the eight available threads. After the chunks are distributed evenly to all threads, there is one chunk left over (iterations 97:100), which executes on thread 0.

**Figure 16** Stride-parallelized loop



**Example**

`prefer_parallel, loop_parallel`

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

```
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 the `J` loop, the load is more evenly balanced 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.



### **critical\_section, end\_critical\_section**

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. Use the `ordered_section` and `end_ordered_section` directives or pragmas for ordered synchronization within a `loop_parallel(ordered)` loop.

A `critical_section` directive or pragma and its associated `end_critical_section` must appear in the same procedure and under the same control flow. They do not have to appear in the same procedure as the parallel construct in which they are used. For instance, the pair can appear in a procedure called from a parallel loop.

The forms of these directives and pragmas are shown in 9.

**Table 34**

**Forms of critical\_section/end\_critical\_section directives and pragmas**

<b>Language</b>	<b>Form</b>
Fortran	<code>C\$DIR CRITICAL_SECTION [ (gate) ]</code>  <code>C\$DIR END_CRITICAL_SECTION</code>
C	<code>#pragma _CNX critical_section [ (gate) ]</code>  <code>#pragma _CNX end_critical_section</code>

The `critical_section` directive/pragma can take an optional `gate` attribute that allows the declaration of multiple critical sections. This is described in “Using gates and barriers” on page 235. Only simple critical sections are discussed in this section.

## Example

### critical\_section

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
```

Because `FUNC` has no side effects and is called in parallel, the `I` loop is 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.

The `LOOP_PARALLEL` directive and the critical section directive 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 did not contain other parallelization inhibitors, the compiler would automatically parallelize the reduction of `SUM` as described in the section “Reductions” on page 108. However, the presence of the call necessitates the `LOOP_PARALLEL` directive, which prevents the compiler from automatically handling the reduction.

This, in turn, requires using either a critical section directive or the `reduction` directive. 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.

In order to justify the cost of the compiler-generated synchronization code associated with the use of critical sections, loops that 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. This helps to determine if parallelization justifies the overhead of the critical section.

For this particular example, the `reduction` directive or `pragma` could have been used in place of the `critical_section`, `end_critical_section` combination. For more information, see the section “Reductions” on page 108.

## Disabling automatic loop thread-parallelization

You can disable automatic loop thread-parallelization by specifying the `+Onoautopar` option on the compiler command line. `+Onoautopar` is only meaningful when specified with the `+Oparallel` option at `+O3` or `+O4`.

This option causes the compiler to parallelize only those loops that are immediately preceded by `prefer_parallel` or `loop_parallel`. Because the compiler does not automatically find parallel tasks or regions, user-specified task and region parallelization is not affected by this option.

---

## Parallelizing tasks

The compiler does not automatically parallelize code outside a loop. However, you can use tasking directives and pragmas to instruct the compiler to parallelize this type of 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 are referred to as a task list. Within a task list, the compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must manually synchronize any dependences between tasks and manually privatize data as necessary.

The forms of these directives and pragmas are shown in Table 35.

**Table 35**

**Forms of task parallelization directives and pragmas**

Language	Form
Fortran	<code>C\$DIR BEGIN_TASKS[ ( <i>attribute-list</i> ) ]</code> <code>C\$DIR NEXT_TASK</code> <code>C\$DIR END_TASKS</code>
C	<code>#pragma _CNX begin_tasks[ ( <i>attribute-list</i> ) ]</code> <code>#pragma _CNX next_task</code> <code>#pragma _CNX end_tasks</code>

where

*attribute-list*

can contain one of the case-insensitive attributes noted in Table 36.

The optional *attribute-list* can contain one of the following attribute combinations, with *m* being an integer constant.

**Table 36**      **Attributes for task parallelization**

Attribute	Description
dist	<p>Instructs the compiler to distribute the tasks across the currently threads, instead of spawning new threads.</p> <p>Use with other valid attributes to <code>begin_tasks</code> inside a <code>parallel/end_parallel</code> region. <code>begin_tasks</code> and <code>parallel/end_parallel</code> must appear inside the same function.</p>
ordered	<p>Causes the tasks to be initiated in their lexical order. That is, the first task in the sequence begins to run on its respective thread before the second and so on.</p> <p>In the absence of the <code>ordered</code> argument, the starting order is indeterminate. While this argument ensures an ordered starting sequence, it does not provide any synchronization between tasks, and does not guarantee any particular ending order.</p> <p>You can manually synchronize the tasks, if necessary, as described in “Parallel synchronization,” on page 233.</p>
max_threads = <i>m</i>	<p>Restricts execution of the specified loop to no more than <i>m</i> threads if specified alone or with the <code>threads</code> attribute. <i>m</i> must be an integer constant.</p> <p><code>max_threads = m</code> is useful when you know the maximum number of threads on which your task runs efficiently.</p> <p>Can include any combination of <code>thread-parallel</code>, <code>ordered</code> or <code>unordered</code> execution.</p>
dist, ordered	<p>Causes ordered invocation of each task across threads, as specified in the attribute list to the <code>parallel</code> directive.</p>

Parallel programming techniques  
Parallelizing tasks

<b>Attribute</b>	<b>Description</b>
dist, max_threads = <i>m</i>	Causes thread-parallelism on no more than <i>m</i> existing threads.
ordered, max_threads = <i>m</i>	Causes ordered parallelism on no more than <i>m</i> threads.
dist, ordered, max_threads = <i>m</i>	Causes ordered thread-parallelism on no more than <i>m</i> existing threads.

**NOTE**

Do not use tasking directives or pragmas unless you have verified that dependences do not exist. You may insert your own synchronization code in the code delimited by the tasking directives or pragmas. The compiler will not perform dependence checking or synchronization on the code in these regions. Synchronization is discussed in "Parallel synchronization," on page 233.

## Example

### Parallelizing tasks

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
```

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

```
C$DIR LOOP_PARALLEL
    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 execute multiple tasks. If there are more threads than tasks, some threads do not execute tasks.

In this example, the `END_TASKS` directive and pragma acts as a barrier. All parallel tasks must complete before the code following `END_TASKS` can execute.

## Example

### Parallelizing tasks

The following C example illustrates how to use these directives to specify simple task-parallelization:

```
#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=0;i<500;i++)
    c[i*2] = d[i];
#pragma _CNX end_tasks
```

In this example, one thread executes the `for` loop, another thread executes the `tsub(x,y)` function call, 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.

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 `for I` loop is inconsequential. There is no way for the loop to run but serially.

The loop induction variable `i` must be manually privatized here because it is used to control loops in two different tasks. If `i` were not private, both tasks would modify it, causing wrong answers. The `task_private` directive and `pragma` is described in detail in the section “`task_private`” on page 218.

Task parallelism can become even more involved, as described in “Parallel synchronization,” on page 233.



---

## Parallelizing regions

A parallel region is a single block of code that is written to run replicated on several threads. Certain scalar code within the parallel region is run by each thread in preparation for work-sharing parallel constructs such as `prefer_parallel(dist)`, `loop_parallel(dist)`, or `begin_tasks(dist)`. The scalar code typically assigns data into `parallel_private` variables so that subsequent references to the data have a high cache hit rate. Within a parallel region, code execution can be restricted to subsets of threads by using conditional blocks that test the thread ID.

Region parallelization differs from task parallelization in that parallel tasks are separate, contiguous blocks of code. When parallelized using the tasking directives and pragmas, each block generally runs on a separate thread. This is in comparison to a single parallel region, which runs on several threads.

Specifying parallel tasks is also typically less time consuming because each thread's work is implicitly defined by the task boundaries. In region parallelization, you must manually modify the region to identify thread-specific code. However, region parallelism can reduce parallelization overhead as discussed in the section explaining the `dist` attribute.

The beginning of a parallel region is denoted by the `parallel` directive or pragma. The end is denoted by the `end_parallel` directive or pragma. `end_parallel` also prevents execution from continuing until all copies of the parallel region have completed.

Within a parallel region, the compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must manually synchronize any dependences between copies of the region and manually privatize data as necessary. In the absence of a `threads` attribute, `parallel` defaults to thread parallelization.

The forms of the regional parallelization directives and pragmas are shown in Table 37.

**Table 37**      **Forms of region parallelization directives and pragmas**

Language	Form
Fortran	C\$DIR PARALLEL[ ( <i>attribute-list</i> ) ]  C\$DIR END_PARALLEL
C	#pragma _CNX parallel( <i>attribute-list</i> )  #pragma _CNX end_parallel

The optional *attribute-list* can contain one of the following attributes (*m* is an integer constant).

**Table 38**      **Attributes for region parallelization**

Attribute	Description
max_threads = <i>m</i>	Restricts execution of the specified region to no more than <i>m</i> threads if specified alone or with the <code>threads</code> attribute. <i>m</i> must be an integer constant.  Can include any combination of ordered, or unordered execution.

**WARNING**      Do not use the parallel region directives or pragmas unless you ensure that dependences do not exist or you insert your own synchronization code, if necessary, in the region. The compiler performs no dependence checking or synchronization on the code delimited by the parallel region directives and pragmas. Synchronization is discussed in “Parallel synchronization,” on page 233.

## Example

### Region parallelization

The following Fortran example provides an implementation of region parallelization using the `PARALLEL` directive:

```

      REAL A(1000,8), B(1000,8), C(1000,8), RDNONLY(1000), SUM(8)
      INTEGER MYTID
      .
      .
      .
C     FIRST INITIALIZATION OF RDNONLY IN SERIAL CODE:
      CALL INIT1(RDNONLY)
      IF(NUM_THREADS() .LT. 8) STOP "NOT ENOUGH THREADS; EXITING"
C$DIR PARALLEL(MAX_THREADS = 8), PARALLEL_PRIVATE(I, J, K, MYTID)
      MYTID = MY_THREAD() + 1 !ADD 1 FOR PROPER SUBSCRIBTING
      DO I = 1, 1000
         A(I, MYTID) = B(I, MYTID) * RDNONLY(I)
      ENDDO
      IF(MYTID .EQ. 1) THEN ! ONLY THREAD 0 EXECUTES SECOND
         CALL INIT2(RDNONLY) ! INITIALIZATION
      ENDIF
      DO J = 1, 1000
         B(J, MYTID) = B(J, MYTID) * RDNONLY(J)
         C(J, MYTID) = A(J, MYTID) * B(J, MYTID)
      ENDDO
      DO K = 1, 1000
         SUM(MYTID) = SUM(MYTID) + A(K,MYTID) + B(K,MYTID) +
C(K,MYTID)
      ENDDO
C$DIR END_PARALLEL

```

In this example, all arrays written to in the parallel code have one dimension for each of the anticipated number of parallel threads. Each thread can work on disjoint data, there is no chance of two threads attempting to update the same element, and, therefore, there is no need for explicit synchronization. The `RDNONLY` array is one-dimensional, but it is never written to by parallel threads. Before the parallel region, `RDNONLY` is initialized in serial code.

The `PARALLEL_PRIVATE` directive is used to privatize the induction variables used in the parallel region. This must be done so that the various threads processing the region do not attempt to write to the same shared induction variables. `PARALLEL_PRIVATE` is covered in more detail in the section “`parallel_private`” on page 220.

At the beginning of the parallel region, the `NUM_THREADS()` intrinsic is called to ensure that the expected number of threads are available. Then the `MY_THREAD()` intrinsic, is called by each thread to determine its thread ID. All subsequent code in the region is executed based on this ID. In the `I` loop, each thread computes one row of `A` using `RDNONLY` and the corresponding row of `B`.

## Parallel programming techniques

### Parallelizing regions

`RONLY` is reinitialized in a subroutine call that is only executed by thread 0 before it is used again in the computation of `B` in the `J` loop. In `J`, each thread computes a row again. The `J` loop similarly computes `C`.

Finally, the `K` loop sums each dimension of `A`, `B`, and `C` into the `SUM` array. No synchronization is necessary here because each thread is running the entire loop serially and assigning into a discrete element of `SUM`.

---

## Reentrant compilation

By default, HP-UX parallel compilers compile for reentrancy in that the compiler itself does not introduce static or global references beyond what exist in the original code. 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 or in C/C++ `static` 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.

---

## Setting thread default stack size

Thread 0's stack can grow to the size specified in the `maxssiz` configurable kernel parameter. Refer to the *Managing Systems and Workgroups* manual for more information on configurable kernel parameters.

Any threads your program spawns (as the result of `loop_parallel` or `tasking` directives or pragmas) receive a default stack size of 80 Mbytes. This means that if the following conditions exist, then you must modify the stack size of the spawned threads using the `CPS_STACK_SIZE` environment variable:

- A parallel construct declares more than 80 Mbytes of `loop_private`, `task_private`, or `parallel_private` data, or
- A subprogram with more than 80 Mbytes of local data is called in `parallel`, or
- The cumulative size of all local variables in a chain of subprograms called in `parallel` exceeds 80 Mbytes,

## Modifying thread stack size

Under `csH`, you can modify the stack size of the spawned threads using the `CPS_STACK_SIZE` environment variable.

The form of the `CPS_STACK_SIZE` environment variable is shown in Table 39.

**Table 39**

**Forms of `CPS_STACK_SIZE` environment variable**

Language	Form
Fortran, C	<code>setenv CPS_STACK_SIZE <i>size_in_kbytes</i></code>

where

*size\_in\_kbytes*

is the desired stack size in kbytes. This value is read at program start-up, and it cannot be changed during execution.

For example, the following command sets the thread stack size to 100 Mbytes:

```
setenv CPS_STACK_SIZE 102400
```

---

## Collecting parallel information

Several 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 variants. They can appear in executable statements anywhere an integer expression is allowed.

The 8-byte functions, which are suffixed with `_8`, are typically only used in Fortran programs in which the default data lengths have been changed using the `-I8` or similar compiler options. When default integer lengths are modified via compiler options in Fortran, the correct intrinsic is automatically chosen regardless of which is specified. These versions expect 8-byte input arguments and return 8-byte values.

### NOTE

All C/C++ code examples presented in this chapter assume that the line below appears above the C code presented. This header file contains the necessary type and function definitions.

```
#include <spp_prog_model.h>
```

## Number of processors

Certain functions return the total number of processors on which the process has initiated threads. These threads are not necessarily active at the time of the call. The forms of these functions are shown in Table 40.

**Table 40**      **Number of processors functions**

<b>Language</b>	<b>Form</b>
Fortran	INTEGER NUM_PROCS()  INTEGER*8 NUM_PROCS_8()
C/C++	int num_procs(void);  long long num_procs_8(void);

`num_procs` is used to dimension automatic and adjustable arrays in Fortran. It may be used in Fortran, C, and C++ to dynamically specify array dimensions and allocate storage.

## Number of threads

Certain functions return the total number of threads the process creates at initiation, regardless of how many are idle or active. The forms of these functions is shown in Table 41.

**Table 41**      **Number of threads functions**

<b>Language</b>	<b>Form</b>
Fortran	INTEGER NUM_THREADS()  INTEGER*8 NUM_THREADS_8()
C/C++	int num_threads(void);  long long num_threads_8(void);

The return value differs from `num_procs` only if threads are oversubscribed.



## Thread ID

When called from parallel code these functions return the spawn thread ID of the calling thread, in the range  $0..N-1$ , 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. The forms of these functions is shown in Table 42.

**Table 42** Thread ID functions

Language	Form
Fortran	<pre>INTEGER MY_THREAD()  INTEGER*8 MY_THREADS_8()</pre>
C/C++	<pre>int my_thread(void);  long long my_thread_8(void);</pre>

When called from serial code, these functions return 0.

## 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. On a single-node system, the thread stack is created in `node_private` memory by default. The forms of these functions is shown in Table 43.

**Table 43** Stack memory type functions

Language	Form
Fortran	<pre>INTEGER MEMORY_TYPE_OF_STACK()  INTEGER*8 MEMORY_TYPE_OF_STACK_8()</pre>
C/C++	<pre>int memory_type_of_stack(void);  long long memory_type_of_stack_8(void);</pre>

Parallel programming techniques  
**Collecting parallel information**

Once HP shared memory classes are assigned, they are implemented throughout your entire program. Very efficient programs are written using these memory classes, as described in “Memory classes,” on page 223. However, these programs also require some manual intervention. Any loops that manipulate variables that are explicitly assigned to a memory class must be manually parallelized. Once a variable is assigned a class, its class cannot change.

This chapter describes the workarounds provided by the HP Fortran 90 and C compilers to support:

- Privatizing loop variables
- Privatizing task variables
- Privatizing region variables

---

## Directives and pragmas for data privatization

This section describes the various directives and pragmas that are implemented to achieve data privatization. These directives and pragmas are discussed in Table 44.

**Table 44** Data Privatization Directives and Pragmas

Directive / Pragma	Description	Level of parallelism
<code>loop_private</code> ( <i>namelist</i> )	Declares a list of variables and/or arrays private to the following loop.	Loop
<code>parallel_private</code> ( <i>namelist</i> )	Declares a list of variables and/or arrays private to the following parallel region.	Region
<code>save_last[(<i>list</i>)]</code>	Specifies that the variables in the comma-delimited list (also named in an associated <code>loop_private</code> ( <i>namelist</i> ) directive or pragma) must have their values saved into the shared variable of the same name at loop termination.	Loop
<code>task_private</code> ( <i>namelist</i> )	Privatizes the variables and arrays specified in <i>namelist</i> for each task specified in the following <code>begin_tasks/end_tasks</code> block.	Task

These directives and pragmas allow you to easily and temporarily privatize parallel loop, task, or region data. When used with `prefer_parallel`, these directives and pragmas do not inhibit automatic compiler optimizations. This facilitates increased performance of your shared-memory program. It occurs with less work than is required when using the standard memory classes for manual parallelization and synchronization.

The data privatization directives and pragmas are used on local variables and arrays of any type, but they should not be used on data assigned to `thread_private`.

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 80 Mbytes in size.

---

## Privatizing loop variables

This section describes the following directives and pragmas associated with privatizing loop variables:

- `loop_private`
- `save_last`

### `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. `loop_private` array dimensions must be identifiable at compile-time.

The compiler assumes that data objects declared to be `loop_private` have no loop-carried dependences with respect to the parallel loops in which they are used. If dependences exist, they must be handled manually using the synchronization directives and techniques described in “Parallel synchronization,” on page 233.

Each parallel thread of execution receives a private copy of the `loop_private` data object for the duration of the loop. No starting values are assumed for the data. Unless a `save_last` directive or pragma is specified, no ending value is assumed. If a `loop_private` data object is referenced within an iteration of the loop, it must be assigned a value previously on that same iteration.

The form of this directive and pragma is shown in Table 45.

**Table 45**

**Form of `loop_private` directive and pragma**

Language	Form
Fortran	<code>C\$DIR LOOP_PRIVATE(<i>namelist</i>)</code>
C	<code>#pragma _CNX loop_private(<i>namelist</i>)</code>

where

*namelist* is a comma-separated list of variables and/or arrays that are to be private to the immediately following loop. *namelist* cannot contain structures, dynamic arrays, allocatable arrays, or automatic arrays.

### Example

#### `loop_private`

The following is a Fortran example of `loop_private`:

```
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
```

A potential loop-carried dependence 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.

If on any iteration none of the `IF` tests pass, an actual LCD exists and privatizing `S` results in wrong answers.

### Example

#### Using `loop_private` with `loop_parallel`

Because the compiler does not automatically perform variable privatization in `loop_parallel` loops, you must manually privatize loop data requiring privatization. This is easily done using the `loop_private` directive or `pragma`.

## Data privatization

### Privatizing loop variables

The following Fortran example shows how `loop_private` manually privatizes loop data:

```
      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 is 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 an LCD exists for `XMFIED` in the `J` loop, but because this loop runs serially on each processor, the dependence is safe.

### 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 customized 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`.



**Example**

**Denoting induction variables in parallel loops**

Consider the following Fortran example:

```
      I = 1
C$DIR LOOP_PARALLEL(IVAR = I)
10    A(I) = ...
      .
      .           ! ASSUME NO DEPENDENCES
      .
      I = I + 1
      IF(I .LE. N) GOTO 10
```

The above is a customized loop that uses `I` as its primary induction variable. To ensure parallelization, the `LOOP_PARALLEL` directive is placed immediately before the start of the loop, and the induction variable, `I`, is specified.

**Example**

**Denoting induction variables in parallel loops**

Primary induction variables in C loops are 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 dependences */
    .
}
}
```

**Secondary induction variables**

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, and they cannot be independent. Secondary induction variables must be assigned `loop_private`.

### Example

#### Secondary induction variables

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!!!
```

In this example,  $J$  does not produce expected values in each iteration because multiple threads are overwriting its value with no synchronization. The compiler cannot privatize  $J$  because it is a loop-carried dependence (LCD). This example is 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
```

As shown in the preceding example,  $J$  is assigned correct values on each iteration because it is a function of  $I$  and is safely privatized.

### Example

#### Secondary induction variables

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 is not recognized.

To manually parallelize this loop, you must remove `j` from the `for` statement, privatize it, and make it a function of `i`.

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.

## **save\_last [ ( *list* ) ]**

A `save_last` directive or pragma causes the thread that executes the last iteration of the loop to write back the private (or local) copy of the variable into the global reference.

The `save_last` directive and pragma allows you to save the final value of `loop_private` data objects assigned in the last iteration of the immediately following loop.

- If *list* (the optional, comma-separated list of `loop_private` data objects) is specified, only the final values of those data objects in *list* are saved.
- If *list* is not specified, the final values of all `loop_private` data objects assigned in the last loop iteration are saved.

The values for this directive and pragma must be assigned in the last iteration. If the assignment is executed conditionally, it is your responsibility to ensure that the condition is met and the assignment executes. Inaccurate results may occur 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 are saved.

The form of this directive and pragma is shown in Table 46.

**Table 46**

**Form of `save_last` directive and pragma**

<b>Language</b>	<b>Form</b>
Fortran	C\$DIR SAVE_LAST [ ( <i>list</i> ) ]
C	#pragma _CNX save_last [ ( <i>list</i> ) ]

`save_last` must appear immediately before or after the associated `loop_private` directive or pragma, or on the same line.

### **Example**

#### **save\_last**

The following is a C example of `save_last`:

```
#pragma _CNX loop_parallel(ivar=i)
#pragma _CNX loop_private(atemp, x, y)
#pragma _CNX save_last(atemp, x)
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;
    x = atemp * a[i];
    y = atemp * c[i];
}
.
.
.
if(atemp > amax) {
.
.
.

```

In this example, the `loop_private` variable `atemp` is conditionally assigned in the loop. In order for `atemp` to be truly private, you 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` pragma ensures that `atemp` and `x` contain the values they are assigned on the last iteration. These values can then be used later in the program. The value of `y`, however, is not available once the loop finishes because `y` is not specified as an argument to `save_last`.

## Example

### `save_last`

There are some loop contexts in which the `save_last` directive and pragma is misleading.

The following Fortran code provides an example of this:

```

C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(S)
C$DIR 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 is saved in this example, you must remember that the `SAVE_LAST` directive applies only to the last (Nth) iteration, with no 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 is predicted, the loop is more efficiently written to exclude the `IF` test, so the presence of a `SAVE_LAST` in such a loop is suspect.

---

## Privatizing task variables

Task privatization is manually specified using the `task_private` directive and `pragma`. `task_private` 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 and `parallel_private` serves for regions.

### `task_private`

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 dependences between the tasks in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in “Parallel synchronization,” on page 233.

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 are assumed for the data. If a `task_private` data object is referenced within a task, it must have been previously assigned a value in that task.

The form of this directive and `pragma` is shown in Table 47.

**Table 47** Form of `task_private` directive and `pragma`

Language	Form
Fortran	<code>C\$DIR TASK_PRIVATE(<i>namelist</i>)</code>
C	<code>#pragma _CNX task_private(<i>namelist</i>)</code>

where

*namelist* is a comma-separated list of variables and/or arrays that are to be private to the immediately following tasks. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

### Example

#### task\_private

The following Fortran code provides an example of task privatization:

```
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
```

In this example, the `WRK` array is used in the first task to temporarily hold the `A` array so that its order is reversed. It serves the same purpose for the `B` array in the second task. `WRK` is assigned before it is used in each task.

---

## Privatizing region variables

Regional privatization is manually specified using the `parallel_private` directive or `pragma`. `parallel_private` is provided to declare a list of variables and/or arrays private to the immediately following parallel region. It serves the same purpose for parallel regions as `task_private` does for tasks, and `loop_private` does for loops.

### `parallel_private`

The `parallel_private` directive must immediately precede, or appear on the same line as, its corresponding `parallel` directive. Using `parallel_private` asserts that there are no dependences in the parallel region.

Do not use `parallel_private` if there are dependences.

Each parallel thread of execution receives a private copy of the `parallel_private` data object for the duration of the region. No starting or ending values are assumed for the data. If a `parallel_private` data object is referenced within a region, it must have been previously assigned a value in the region.

The form of this directive and `pragma` is shown in Table 48.

**Table 48**

**Form of `parallel_private` directive and `pragma`**

Language	Form
Fortran	C\$DIR PARALLEL_PRIVATE( <i>namelist</i> )
C	#pragma _CNX parallel_private( <i>namelist</i> )

where

*namelist* is a comma-separated list of variables and/or arrays that are to be private to the immediately following parallel region. *namelist* cannot contain dynamic, allocatable, or automatic arrays.



**Example**            `parallel_private`

The following Fortran code shows how `parallel_private` privatizes regions:

```

REAL A(1000,8), B(1000,8), C(1000,8), AWORK(1000), SUM(8)
INTEGER MYTID
.
.
.
C$DIR PARALLEL(MAX_THREADS = 8)
C$DIR PARALLEL_PRIVATE(I,J,K,L,M,AWORK,MYTID)
IF(NUM_THREADS() .LT. 8) STOP "NOT ENOUGH THREADS; EXITING"
MYTID = MY_THREAD() + 1 !ADD 1 FOR PROPER SUBSCRIBTING
DO I = 1, 1000
  AWORK(I) = A(I, MYTID)
ENDDO
DO J = 1, 1000
  A(J, MYTID) = AWORK(J) + B(J, MYTID)
ENDDO
DO K = 1, 1000
  B(K, MYTID) = B(K, MYTID) * AWORK(K)
  C(K, MYTID) = A(K, MYTID) * B(K, MYTID)
ENDDO
DO L = 1, 1000
  SUM(MYTID) = SUM(MYTID) + A(L,MYTID) + B(L,MYTID) + C(L,MYTID)
ENDDO
DO M = 1, 1000
  A(M, MYTID) = AWORK(M)
ENDDO
C$DIR END_PARALLEL

```

This example is similar to the example on page 197 in the way it checks for a certain number of threads and divides up the work among those threads. The example additionally introduces the `parallel_private` variable `AWORK`.

Each thread initializes its private copy of `AWORK` to the values contained in a dimension of the array `A` at the beginning of the parallel region. This allows the threads to reference `AWORK` without regard to thread ID. This is because no thread can access any other thread's copy of `AWORK`. Because `AWORK` cannot carry values into or out of the region, it must be initialized within the region.

### **Induction variables in region privatization**

All induction variables contained in a parallel region must be privatized. Code contained in the region runs on all available threads. Failing to privatize an induction variable would allow each thread to update the same shared variable, creating indeterminate loop counts on every thread.

In the previous example, in the  $J$  loop, after `AWORK` is initialized, `AWORK` is effectively used in a reduction on `A`; at this point its contents are identical to the `MYTID` dimension of `A`. After `A` is modified and used in the `K` and `L` loops, each thread restores a dimension of `A`'s original values from its private copy of `AWORK`. This carries the appropriate dimension through the region unaltered.

The V-Class server implements only one partition of hypernode-local memory. This is accessed using the `thread_private` and `node_private` virtual memory classes. This chapter includes discussion of the following topics:

- Private versus shared memory
- Memory class assignments

The information in this chapter is provided for programmers who want to manually optimize their shared-memory programs on a single-node server. This is ultimately achieved by using compiler directives or pragmas to partition memory and otherwise control compiler optimizations. It can also be achieved using storage class specifiers in C and C++.

## Porting multinode applications to single-node servers

Programs developed to run on multinode servers, such as the legacy X-Class server, can be run on K-Class or V-Class servers. The program runs as it would on one node of a multinode machine.

When a multinode application is executed on a single-node server:

- All `PARALLEL`, `LOOP_PARALLEL`, `PREFER_PARALLEL`, and `BEGIN_TASKS` directives containing `node` attributes are ignored.
- All variables, arrays and pointers that are declared to be `NEAR_SHARED`, `FAR_SHARED`, or `BLOCK_SHARED` are assigned to the `NODE_PRIVATE` class.
- The `THREAD_PRIVATE` and `NODE_PRIVATE` classes remain unchanged and function as usual.

See the *Exemplar Programming Guide for HP-UX Systems* for a complete description of how to program multinode applications using HP parallel directives.

## Private versus shared memory

Private and shared data are differentiated by their accessibility and by the physical memory classes in which they are stored.

`thread_private` data is stored in node-local memory. Access to `thread_private` is restricted to the declaring thread.

When porting multinode applications to the HP single-node machine, all legacy shared memory classes (such as `near_shared`, `far_shared`, and `block_shared`) are automatically mapped to the `node_private` memory class. This is the default memory class on the K-Class and V-Class servers.

### `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. This virtual address maps to unique physical addresses in hypernode-local physical memory.

Any sharing of `thread_private` data items between threads (regardless of whether they are running on the same node) must be done by synchronized copying of the item into a shared variable, or by message passing.

#### NOTE

`thread_private` data cannot be initialized in C, C++, or in Fortran `DATA` statements.

### `node_private`

`node_private` data is shared among the threads of a process running on a given node. It is the default memory class on the V-Class single-node server, and does not need to be explicitly specified. `node_private` data items have one virtual address, and any thread on a node can access that node's `node_private` data using the same virtual address. This virtual address maps to a unique physical address in node-local memory.

## Memory class assignments

In Fortran, compiler directives are used to assign memory classes to data items. In C and C++, memory classes are assigned through the use of syntax extensions, which are defined in the header file `/usr/include/spp_prog_model.h`. This file must be included in any C or C++ program that uses memory classes. In C++, you can also use operator `new` to assign memory classes.

- The Fortran memory class declarations must appear with other specification statements; they cannot appear within executable statements.
- In C and C++, parallel storage class extensions are used, so memory classes are assigned in variable declarations.

On a single-node system, HP compilers provide mechanisms for statically assigning memory classes. This chapter discusses these memory class assignments.

The form of the directives and pragmas associated with is shown in Table 49.

**Table 49** Form of memory class directives and variable declarations

Language	Form
Fortran	<code>C\$DIR <i>memory_class_name</i>(<i>namelist</i>)</code>
C/C++	<code>#include &lt;spp_prog_model.h&gt;</code> . . . <code>[<i>storage_class_specifier</i>] <i>memory_class_name</i> <i>type_specifier</i> <i>namelist</i></code>

where (for Fortran)

*memory\_class\_name*

can be `THREAD_PRIVATE`, or `NODE_PRIVATE`

*namelist*

is a comma-separated list of variables, arrays, and/or COMMON block names to be assigned the class *memory\_class\_name*. COMMON block names must be enclosed in slashes (/), and only entire COMMON blocks can be assigned a class. This means arrays and variables in *namelist* must not also appear in a COMMON block and must not be equivalenced to data objects in COMMON blocks.

where (for C)

*storage\_class\_specifier*

specifies a nonautomatic storage class

*memory\_class\_name*

is the desired memory class (thread\_private, node\_private)

*type\_specifier*

is a C or C++ data type (int, float, etc.)

*namelist*

is a comma-separated list of variables and/or arrays of type *type\_specifier*

## C and C++ data objects

In C and C++, data objects that 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; attempting to do so results in a compile-time error.

Data objects declared at file scope and assigned a memory class need not specify a storage class.

All C and C++ code examples presented in this chapter assume that the following line appears above the code presented:

```
#include <spp_prog_model.h>
```

This header file maps user symbols to the implementation reserved space.

## Memory classes

### Memory class assignments

If operator `new` is used, it is also assumed that the line below appears above the code:

```
#include <new.h>
```

If you assign a memory class to a C or C++ structure, all structure members must be of the same class.

Once a data item is assigned a memory class, the class cannot be changed.

## 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` and `node_private` classes. Static assignments for all classes are explained in the subsections that follow.

### `thread_private`

Because `thread_private` variables are replicated for every thread, static declarations make the most sense for them.

#### Example

### `thread_private`

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 scalar 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.



**Example**

**thread\_private**

The following C/C++ example demonstrates several ways to declare `thread_private` storage. 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;  
    .  
    .  
    .  
}
```

The C/C++ 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 file scope in another file that is linked with this one.

**Example**

**thread\_private COMMON blocks in parallel subroutines**

Data local to a procedure that is called in parallel is effectively private because storage for it is allocated on the thread's private stack. However, if the data is in a Fortran `COMMON` block (or if it appears in a `DATA` or `SAVE` statement), it is not stored on the stack. Parallel accesses to such nonprivate data must be synchronized if it is assigned a shared class. Additionally, if the parallel copies of the procedure do not need to share the data, it can be assigned a private class.

Memory classes  
Memory class assignments

Consider the following Fortran example:

```
      INTEGER A(1000,1000)
      .
      .
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, N
        CALL PARCOM(A(1,I))
        .
        .
      ENDDO
      SUBROUTINE PARCOM(A)
      INTEGER A(*)
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      INTEGER TEMP1, TEMP2
      D(1:1000) = ...
      .
      .
      CALL PARCOM2(A, JTA)
      .
      .
      END

      SUBROUTINE PARCOM2(B,JTA)
      INTEGER B(*), JTA
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      DO J = 1, 1000
        C(J) = D(J) * B(J)
      ENDDO
      END
      .
      .
      .
```

In this example, COMMON block BLK1 is declared THREAD\_PRIVATE, so every parallel instance of PARCOM gets its own copy of the arrays C and D.

Because this code is already thread-parallel when the COMMON block is defined, no further parallelism is possible, and BLK1 is therefore suitable for use anywhere in PARCOM. The local variables TEMP1 and TEMP2 are allocated on the stack, so each thread effectively has private copies of them.

## node\_private

Because the space for `node_private` variables is physically replicated, 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.

## Example

### node\_private

The following example shows several ways to declare `node_private` data objects in C and 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/C++ example.

Memory classes  
Memory class assignments

Most of the manual parallelization techniques discussed in “Parallel programming techniques,” on page 175, allow you to take advantage of the compilers’ automatic dependence checking and data privatization. The examples that used the `LOOP_PRIVATE` and `TASK_PRIVATE` directives and pragmas in “Data privatization,” on page 207, are exceptions to this. In these cases, manual privatization is required, but is performed on a loop-by-loop basis. Only the simplest data dependences are handled.

This chapter discusses manual parallelizations and that handle multiple and ordered data dependences. This includes a discussion of the following topics:

- Thread-parallelism
- Synchronization tools
- Synchronizing code

## Thread-parallelism

Only one level of parallelism is supported: thread-parallelism. If you attempt to spawn thread-parallelism from within a thread-parallel, your directives on the inner thread-parallel construct are ignored.

### Thread ID assignments

Programs are initiated as a collection of threads, one per available processor. All but thread 0 are idle until parallelism is encountered.

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 returns 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 is changed using various compiler directives.

If the parallel structure is thread-parallel, then `num_threads()` threads are 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 range from 0 to one less than your limit.

## Synchronization tools

The compiler cannot automatically parallelize loops containing complex dependences. However, a rich set of directives, pragmas, and data types is available to help you manually parallelize such loops by synchronizing and ordering access to the code containing the dependence.

These directives can also be used to synchronize dependences in parallel tasks. They allow you to efficiently exploit parallelism in structures that would otherwise be unparallelizable.

### Using gates and barriers

Gates allow you to restrict execution of a block of code to a single thread. They are allocated, locked, unlocked, and deallocated using the functions described in “Synchronization functions” on page 237. They can also 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 and then thread 0 can proceed past the barrier.

Gates and barriers use dynamically allocatable variables, declared using compiler directives in Fortran and using data declarations in C and C++. They may be initialized and referenced only by passing them as arguments to the functions discussed in the following sections.

The forms of these variable declarations are shown in Table 50.

**Table 50**

**Forms of gate and barriers variable declarations**

Language	Form
Fortran	C\$DIR GATE( <i>namelist</i> ) C\$DIR BARRIER( <i>namelist</i> )
C/C++	gate_t <i>namelist</i> ; barrier_t <i>namelist</i> ;

where

*namelist* is a comma-separated list of one or more gate or barrier names, as appropriate.

### **In C and C++**

In C and C++, gates and barriers should appear only in definition and declaration statements, and as formal, and actual arguments. They declare default-size variables.

### **In Fortran**

The Fortran `gate` and `barrier` variable declarations can only appear:

- In `COMMON` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In `DIMENSION` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In preceding type statements
- As dummy arguments
- As actual arguments

Gate and barrier types override other same-named types declared prior to the gate/barrier pragmas. Once a variable is defined 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/`BARRIER` directive. The `COMMON` block should contain only gates or only barriers. Arrays of gates or barriers must be dimensioned using `DIMENSION` statements. The `DIMENSION` statement must precede the `GATE` directive/`BARRIER` directive.



## Synchronization functions

The Fortran, C, and C++ allocation, deallocation, lock and unlock functions for use with gates and barriers are described in this section. The 4- and 8-byte versions are provided. The 8-byte Fortran functions are primarily for use with compiler options that change the default data size to 8 bytes (for example, `-I8`). 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 throughout this section.

### Allocation functions

Allocation functions allocate memory for a gate or barrier. When first allocated, gate variables are unlocked. The forms of these allocation functions are shown in Table 51.

**Table 51**

**Forms of allocation functions**

Language	Form
Fortran	<pre>INTEGER FUNCTION ALLOC_GATE(<i>gate</i>) INTEGER FUNCTION ALLOC_BARRIER(<i>barrier</i>)</pre>
C/C++	<pre>int alloc_gate(gate_t *<i>gate_p</i>); int alloc_barrier(barrier_t *<i>barrier_p</i>);</pre>

where (in Fortran)

*gate* and *barrier*

are gate or barrier variables.

where (in C/C++)

*gate\_p* and *barrier\_p*

are pointers of the indicated type.

## Deallocation functions

The deallocation functions free the memory assigned to the specified gate or barrier variable. The forms of these deallocation functions are shown in Table 52.

**Table 52**      **Forms of deallocation functions**

Language	Form
Fortran	INTEGER FUNCTION FREE_GATE( <i>gate</i> )  INTEGER FUNCTION FREE_BARRIER( <i>barrier</i> )
C/C++	int free_gate(gate_t * <i>gate_p</i> );  int free_barrier(barrier_t * <i>barrier_p</i> );

where (in Fortran)

*gate* and *barrier*

are gate or barrier variables previously declared in the gate and barrier allocation functions.

where (in C/C++)

*gate\_p* and *barrier\_p*

are pointers of the indicated type.

**NOTE**      Always free gates and barriers after using them.

## Locking functions

The locking 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 only if a wait is not required. If the gate is acquired, the functions return 0; if not, they return -1.

The forms of these locking functions are shown in Table 53.

**Table 53**      **Forms of locking functions**

Language	Form
Fortran	<pre>INTEGER FUNCTION LOCK_GATE(<i>gate</i>) INTEGER FUNCTION COND_LOCK_GATE(<i>gate</i>)</pre>
C/C++	<pre>int lock_gate(gate_t *<i>gate_p</i>); int cond_lock_gate(gate_t *<i>gate_p</i>);</pre>

where (in Fortran)

*gate*              is a gate variable.

where (in C/C++)

*gate\_p*            is a pointer of the indicated type.

## Unlocking functions

The unlocking 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 that case it might be unlocked by a single different thread in a parallel construct.

The form of these unlocking functions is shown in Table 54.

**Table 54** Form of unlocking functions

Language	Form
Fortran	INTEGER FUNCTION UNLOCK_GATE( <i>gate</i> )
C/C++	int unlock_gate(gate_t * <i>gate_p</i> );

where (in Fortran)

*gate* is a gate variable.

where (in C/C++)

*gate\_p* is a pointer of the indicated type.

## Wait functions

The wait functions use a barrier to cause the calling thread to wait until the specified number of threads call the function. At this point all threads are released from the function simultaneously.

The form of the wait functions is shown in Table 55.

**Table 55**      **Form of wait functions**

Language	Form
Fortran	INTEGER FUNCTION WAIT_BARRIER( <i>barrier</i> , <i>nthr</i> )
C/C++	int wait_barrier(barrier_t * <i>barrier_p</i> , const int * <i>nthr</i> );

where (in Fortran)

*barrier*            is a barrier variable of the indicated type and *nthr* is the number of threads calling the routine.

where (in C/C++)

*barrier\_p*          is a pointer of the indicated type and *nthr* is a pointer referencing the number of threads calling the routine.

You can use a barrier variable in multiple calls to the `wait` function, if you ensure that two such barriers are not simultaneously active. You must also verify that *nthr* reflects the correct number of threads.

## **sync\_routine**

Among the most basic optimizations performed by the HP compilers is code motion, which is described in “Standard optimization features,” on page 35. This optimization moves code across routine calls. If the routine 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 may result in wrong answers.

The compiler is aware of all synchronization functions and does not move code across them when they appear directly in code. However, if the synchronization function is hidden in a user-defined routine, the compiler has no way of knowing about it and may move code across it.

Any time you call synchronization functions indirectly using your own routines, you must identify your routines with a `sync_routine` directive or pragma.

The form of `sync_routine` is shown in Table 56.

**Table 56**

**Form of `sync_routine` directive and pragma**

<b>Language</b>	<b>Form</b>
Fortran	C\$DIR SYNC_ROUTINE ( <i>routinelist</i> )
C	#pragma CNX sync_routine ( <i>routinelist</i> )

where

*routinelist* is a comma-separated list of synchronization routines.

## Example

### sync\_routine

`sync_routine` is effective only for the listed routines that lexically follow it in the same file where it appears. The following Fortran code example features the `sync_routine` directive:

```

      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

```

In this example, `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`.

Programming techniques such as this are used to implement portable code across several parallel architectures that support critical sections. This would be done using different syntax. For example, `MY_LOCK` and `MY_UNLOCK` could simply be modified to call the correct locking and unlocking functions.

## Example

### sync\_routine

The following C example achieves the same task as shown in the previous Fortran example:

```
#include <spp_prog_model.h>
main() {
    int i, n, lck, sum, a[1000];
    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;
}
```



## `loop_parallel(ordered)`

The `loop_parallel(ordered)` directive and pragma is designed to be used with ordered sections to execute loops with ordered dependences 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 dependences using the ordered section directives, pragmas, or the synchronization intrinsics (see “Critical sections” on page 247 of this chapter for more information).

### Example

#### `loop_parallel, ordered`

The following Fortran code shows how `loop_parallel(ordered)` is structured:

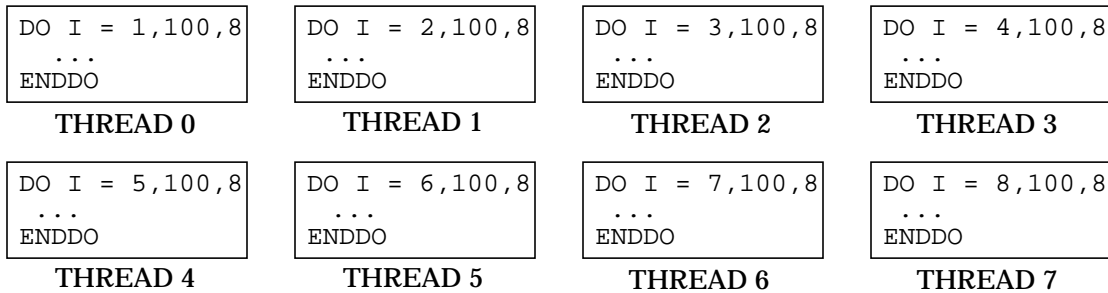
```
C$DIR LOOP_PARALLEL(ORDERED)
DO I = 1, 100
  .
  . !CODE CONTAINING ORDERED SECTION
  .
ENDDO
```

Assume that the body of this loop contains code that is parallelizable except for an ordered data dependence (otherwise there is no need to order the parallelization). 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
```

Figure 17 illustrates this assumption.

**Figure 17**      **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 prevents one thread from executing the code in the ordered section until the previous thread exits the section. This means that thread 0 cannot enter the section for iteration 9 until thread 7 exits it for iteration 8.

This is efficient only 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 shown in “Synchronizing code” on page 250.

## Critical sections

Critical sections allow you to synchronize simple, nonordered dependences. You must use the `critical_section` directive or `pragma` to enter a critical section, and the `end_critical_section` directive or `pragma` to exit one.

Critical sections 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. This means that the directives can exist in a procedure that is called in parallel.

The forms of these directives and pragmas are shown in Table 57.

**Table 57** **Forms of `critical_section`, `end_critical_section` directives and pragmas**

Language	Form
Fortran	<pre>C\$DIR CRITICAL_SECTION[ (<i>gate</i>) ] ... C\$DIR END_CRITICAL_SECTION</pre>
C	<pre>#pragma _CNX critical_section[ (<i>gate</i>) ] ... #pragma _CNX end_critical_section</pre>

where

*gate* is an optional gate variable used for access to the critical section. *gate* must be appropriately declared as described in the “Using gates and barriers” on page 235.

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) 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.

**NOTE**

Critical sections 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 dependence.

## Ordered sections

Ordered sections allow you to synchronize dependences 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.

The forms of these directives and pragmas are shown in Table 58.

**Table 58**

**Forms of `ordered_section`, `end_ordered_section` directives and pragmas**

Language	Form
Fortran	<code>C\$DIR ORDERED_SECTION(<i>gate</i>)</code> ... <code>C\$DIR END_ORDERED_SECTION</code>
C	<code>#pragma _CNX ordered_section(<i>gate</i>)</code> ... <code>#pragma _CNX end_ordered_section</code>

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 “Using gates and barriers” section of this chapter.

Ordered sections must be entered through `ordered_section` and exited through `end_ordered_section`. They cannot contain branches to outside the section. Ordered sections are subject to the same control flow rules as critical sections.

**NOTE**

As with critical sections, ordered sections should be used 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 dependence.

## Synchronizing code

Code containing dependences are parallelized by synchronizing the way the parallel tasks access the dependence. This is done manually using the gates, barriers and synchronization functions discussed earlier in this chapter, or semiautomatically using critical and ordered sections, described in the following sections.

### Using critical sections

The `critical_section` example on page 190 isolates a single critical section in a loop, so that 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 dependences and dependences in manually-defined parallel tasks are handled when user-defined gates are used with the directives.

#### Example

#### critical sections

The following Fortran example, however, uses the manual methods of code synchronization:

```
      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
```

```
CONTRIB2 = 0.0
DO I = 1, N
  CONTRIB2 = CONTRIB2 + FUNC2(J)
ENDDO
.
.
.
C$DIR CRITICAL_SECTION (SUM_GATE)
  GLOBAL_SUM = GLOBAL_SUM + CONTRIB2
C$DIR END_CRITICAL_SECTION
.
.
.
C$DIR END_TASKS
  LOCK = FREE_GATE(SUM_GATE)
```

Here, both parallel tasks must access the shared GLOBAL\_SUM variable. To ensure that GLOBAL\_SUM is updated by only 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. Because access to both critical sections is controlled by a single gate, the sections must execute one at a time.

## Example

### Gated critical sections

Gated critical sections are also useful in loops containing multiple critical sections when there are dependences between the critical sections. If no dependences exist between the sections, gates are not needed. The compiler automatically supplies a unique gate for every critical section lacking a gate.

The C example below uses gates so that threads do not update at the same time, within a critical section:

```
static far_shared float absum;
static gate_t gatel;
int adjb[...];
.
.
.
lock = alloc_gate(&gatel);
#pragma _CNX loop_parallel(ivar=i)
for(i=0;i<n;i++) {
    a[i] = b[i] + c[i];
#pragma _CNX critical_section(gatel)
    absum = absum + a[i];
#pragma _CNX end_critical_section
    if(adjb[i]) {
        b[i] = c[i] + d[i];
#pragma _CNX critical_section(gatel)
        absum = absum + b[i];
#pragma _CNX end_critical_section
    }
    .
    .
}
lock = free_gate(&gatel);
```

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 chooses unique gates for each section, potentially resulting in incorrect answers. There must be a substantial amount of parallelizable code outside of these critical sections to make parallelizing this loop cost-effective.



## Using ordered sections

Like critical sections, ordered sections lock and unlock 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.

You must use a `loop_parallel(ordered)` directive or pragma to parallelize any loop containing an ordered section. See “`loop_parallel(ordered)`” on page 245 for a description of this.

### Example

#### Ordered sections

The following Fortran example 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
```

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, the dependence is isolated. The loop is parallelized as shown below:

```
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)
```

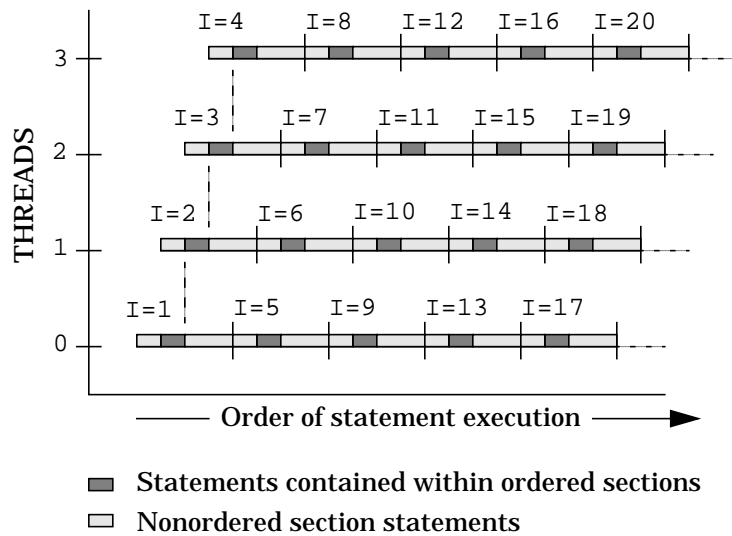
Parallel synchronization  
Synchronizing code

```
C$DIR  END_ORDERED_SECTION
      . ! MORE PARALLELIZABLE CODE...
      .
      .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

The ordered section containing the  $A(I)$  assignment executes in iteration order. This ensures that the value of  $A(I-1)$  used in the assignment is always valid. Assuming this loop runs on four threads, the synchronization of statement execution between threads is illustrated in Figure 18.

**Figure 18**

**LOOP\_PARALLEL(ORDERED) synchronization**



As shown by the dashed lines between initial iterations for each thread, one ordered section must be completed before the next is allowed to begin execution. Once a thread exits an ordered section, it cannot reenter 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. Only brief idle periods occur on 1, 2, and 3 at the beginning of the loop, and on 0, 1, and 2 at the end.

## Example

### Ordered section limitations

Each thread in a parallel loop containing an ordered section must pass through the ordered section exactly 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 IS THE BLANK ORDERED SECTION:
C$DIR   ORDERED_SECTION(LCD)
C$DIR   END_ORDERED_SECTION
ENDIF
.
.
.
ENDDO
LOCK = FREE_GATE(LCD)

```

No matter which path through the IF statement the loop takes, and though the ELSE section is empty, it must pass through the ordered section. This allows the compiler to properly synchronize the ordered loop. It is assumed that a substantial amount of parallel code exists outside the ordered sections, to offset the synchronization overhead.

## Example

### Ordered section limitations

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 better understand this:

- 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 cannot 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.
- For  $I > 96$ , all 6 threads are no longer needed. In a single loop nest this would not pose a problem as 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. As a result, deadlock occurs.

To handle this problem, you can expand the ordered section to include the entire  $j$  loop, as shown in the following C example:

```
#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
}
```

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.

Another approach is to manually interchange the  $i$  and  $j$  loops, as shown in the following Fortran example:

```
      DO J = 1, M
        LOCK = UNLOCK_GATE(ORDGATE)
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. The ordered section is not isolated from its parent loop, so the loop can terminate normally. This example has added benefit; elements of  $A$  are accessed more efficiently.

## Manual synchronization

Ordered and critical sections allow you to isolate dependences in a structured, semiautomatic manner. The same isolation is accomplished manually using the functions discussed in “Synchronization functions” on page 237.

### Example

#### Critical sections and gates

Below is a simple critical section Fortran example using `loop_parallel`:

```
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 is manually implemented in Fortran, using gate functions, 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`.

## Example **Conditionally lock critical sections**

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.

This situation is illustrated in the following Fortran example. Task 1 performs I/O for the 7 other tasks, which perform parallel computations by calling the `THREAD_WRK` subroutine:

```

COMMON INGATE,OUTGATE,COMPBAR
C$DIR GATE (INGATE, OUTGATE)
C$DIR BARRIER (COMPBAR)
REAL DIN(:), DOUT(:)      ! I/O BUFFERS FOR TASK 1
ALLOCATABLE DIN, DOUT     ! THREAD 0 WILL ALLOCATE
REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
INTEGER NIN/0/,NOUT/0/ ! QUEUE ENTRY COUNTERS
C   CIRCULAR BUFFER POINTERS:
INTEGER IN_QIN/1/,OUT_QIN/1/,IN_QOUT/1/,OUT_QOUT/1/
COMMON /DONE/ DONEIN, DONECOMP
LOGICAL DONECOMP, DONEIN
C   SIGNALS FOR COMPUTATION DONE AND INPUT DONE
LOGICAL COMPDONE, INDONE
C   FUNCTIONS TO RETURN DONECOMP AND DONEIN
LOGICAL INFLAG, OUTFLAG ! INPUT READ AND OUTPUT WRITE FLAGS
C$DIR THREAD_PRIVATE (INFLAG,OUTFLAG) ! ONLY NEEDED BY TASK 1
C   (WHICH RUNS ON THREAD 0)
IF (NUM_THREADS() .LT. 8) STOP 1
IN = 10
OUT = 11
LOCK = ALLOC_GATE(INGATE)
LOCK = ALLOC_GATE(OUTGATE)
IBAR = ALLOC_BARRIER(COMPBAR)
DONECOMP = .FALSE.
C$DIR BEGIN_TASKS          ! TASK 1 STARTS HERE
INFLAG = .TRUE.
DONEIN = .FALSE.
ALLOCATE(DIN(1000),DOUT(1000)) ! ALLOCATE LOCAL BUFFERS
DO WHILE(.NOT. INDONE() .OR. .NOT. COMPDONE() .OR. NOUT .GT. 0)
C   DO TILL EOF AND COMPUTATION DONE AND OUTPUT DONE
IF(NIN.LT.1000.AND.(.NOT.COMPDONE()) .AND.(.NOT. INDONE())) THEN

```

## Parallel synchronization Synchronizing code

```

C           FILL QUEUE
IF (INFLAG) THEN ! FILL BUFFER FIRST:
  READ(IN, IOSTAT = IOS) DIN ! READ A RECORD; QUIT ON EOF
  IF(IOS .EQ. -1) THEN
    DONEIN = .TRUE. ! SIGNAL THAT INPUT IS DONE
    INFLAG = .TRUE.
  ELSE
    INFLAG = .FALSE.
  ENDIF
ENDIF
C SYNCHRONOUSLY ENTER INTO INPUT QUEUE:
C BLOCK QUEUE ACCESS WITH INGATE:
IF (COND_LOCK_GATE(INGATE) .EQ. 0 .AND. .NOT. INDONE()) THEN
  QIN(:,IN_QIN) = DIN(:) ! COPY INPUT BUFFER INTO QIN
  IN_QIN=1+MOD(IN_QIN,1000) ! INCREMENT INPUT BUFFER PTR
  NIN = NIN + 1 ! INCREMENT INPUT QUEUE ENTRY COUNTER
  INFLAG = .TRUE.
  LOCK = UNLOCK_GATE(INGATE) ! ALLOW INPUT QUEUE ACCESS
ENDIF
ENDIF
C SYNCHRONOUSLY REMOVE FROM OUTPUT QUEUE:
C BLOCK QUEUE ACCESS WITH OUTGATE:
IF (COND_LOCK_GATE(OUTGATE) .EQ. 0) THEN
  IF (NOUT .GT. 0) THEN
    DOUT(:)=QOUT(:,OUT_QOUT) ! COPY OUTPUT QUE INTO BUFR
    OUT_QOUT=1+MOD(OUT_QOUT,1000)
  C INCREMENT OUTPUT BUFR PTR
    NOUT = NOUT - 1 ! DECREMENT OUTPUT QUEUE ENTRY COUNTR
    OUTFLAG = .TRUE.
  ELSE
    OUTFLAG = .FALSE.
  ENDIF
  LOCK = UNLOCK_GATE(OUTGATE)
C ALLOW OUTPUT QUEUE ACCESS
  IF (OUTFLAG) WRITE(OUT) DOUT ! WRITE A RECORD
ENDIF
ENDDO
C           TASK 1 ENDS HERE
C$DIR NEXT_TASK           ! TASK 2:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK           ! TASK 3:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK           ! TASK 4:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK           ! TASK 5:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK           ! TASK 6:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK           ! TASK 7:
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
IBAR = WAIT_BARRIER(COMPBAR,7)

```



```
C$DIR NEXT_TASK                ! TASK 8:  
CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)  
IBAR = WAIT_BARRIER(COMPBAR,7)  
DONECOMP = .TRUE.  
C$DIR END_TASKS  
END
```

Before looking at the `THREAD_WRK` subroutine it is necessary to examine these parallel tasks, particularly task 1, the I/O server. Task 1 performs all the I/O required by all the tasks:

- Conditionally locked gates control task 1's access to one section of code that fills the input queue and one that empties the output queue.
- Task 1 works by first filling an input buffer. The code that does this does not require gate protection because no other tasks attempt to access the input buffer array.
- The section of code where the input buffer is copied into the input queue, however, must be protected by gates to prevent any threads from trying to read the input queue while it is being filled.

The other seven tasks perform computational work, receiving their input from and sending their output to task 1's queues. If a task acquires a lock on the input queue, task 1 cannot fill it until the task is done reading from it.

- When task 1 cannot get a lock to access the input queue code, it tries to lock the output queue code.
- If it gets a lock here, it can copy the output queue into the output buffer array and relinquish the lock. It can then proceed to empty the output buffer.
- If another task is writing to the output queue, task 1 loops back and begins the entire process over again.
- When the end of the input file is reached, all computation is complete, and the output queue is empty: task 1 is finished.

#### NOTE

The task loops on `DONEIN` (using `INDONE( )`), which is initially false. When input is exhausted, `DONEIN` is set to true, signalling all tasks that there is no more input.

The `INDONE( )` function references `DONEIN`, forcing a memory reference. If `DONEIN` were referenced directly, the compiler might optimize it into a register and consequently not detect a change in its value.

## Parallel synchronization

### Synchronizing code

This means that task 1 has four main jobs to do:

- 1 Read input into input buffer—no other tasks access the input buffer. This is done in parallel regardless of what other tasks are doing, as long as the buffer needs filling.
- 2 Copy input buffer into input queue—the other tasks read their input from the input queue, therefore it can only be filled when no computational task is reading it. This section of code is protected by the `INGATE` gate. It can run in parallel with the computational portions of other tasks, but only one task can access the input queue at a time.
- 3 Copy output queue into output buffer—the output queue is where other tasks write their output. It can only be emptied when no computational task is writing to it. This section of code is protected by the `OUTGATE` gate. It can run in parallel with the computational portions of other tasks, but only one task can access the output queue at a time.
- 4 Write out output buffer—no other tasks access the output buffer. This is done in parallel regardless of what the other tasks are doing.

Next, it is important to look at the subroutine `THREAD_WRK`, which tasks 2-7 call to perform computations.

```
SUBROUTINE
> THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
  INTEGER NIN,NOUT
  REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
  INTEGER OUT_QIN, OUT_QOUT
  COMMON INGATE,OUTGATE,COMPBAR
C$DIR GATE(INGATE, OUTGATE)
  REAL WORK(1000) ! LOCAL THREAD PRIVATE WORK ARRAY
  LOGICAL OUTFLAG, INDONE
  OUTFLAG = .FALSE.
C$DIR THREAD_PRIVATE (WORK) ! EVERY THREAD WILL CREATE A COPY

  DO WHILE(.NOT. INDONE() .OR. NIN.GT.0 .OR. OUTFLAG)
C
      WORK/QOUT EMPTYING LOOP
      IF (.NOT. OUTFLAG) THEN ! IF NO PENDING OUTPUT
C$DIR CRITICAL_SECTION (INGATE) ! BLOCK ACCESS TO INPUT QUE
          IF (NIN .GT. 0) THEN ! MORE WORK TO DO
              WORK(:) = QIN(:,OUT_QIN)
              OUT_QIN = 1 + MOD(OUT_QIN, 1000)
              NIN = NIN - 1
              OUTFLAG = .TRUE.
C
          INDICATE THAT INPUT DATA HAS BEEN RECEIVED
      ENDIF
C$DIR END_CRITICAL_SECTION
  .
```

```

. ! SIGNIFICANT PARALLEL CODE HERE USING WORK ARRAY
.
ENDIF
IF (OUTFLAG) THEN ! IF PENDING OUTPUT, MOVE TO OUTPUT QUEUE
C AFTER INPUT QUEUE IS USED IN COMPUTATION, FILL OUTPUT QUEUE:
C$DIR CRITICAL_SECTION (OUTGATE) ! BLOCK ACCESS TO OUTPUT QUEUE
IF(NOUT.LT.1000) THEN
C IF THERE IS ROOM IN THE OUTPUT QUEUE
QOUT(:,IN_QOUT) = WORK(:) ! COPY WORK INTO OUTPUT QUEUE
IN_QOUT =1+MOD(IN_QOUT,1000) ! INCREMENT BUFFER PTR
NOUT = NOUT + 1 ! INCREMENT OUTPUT QUEUE ENTRY COUNTER
OUTFLAG = .FALSE. ! INDICATE NO OUTPUT PENDING
ENDIF
C$DIR END_CRITICAL_SECTION
ENDIF
ENDDO ! END WORK/QOUT EMPTYING LOOP
END ! END THREAD_WRK

LOGICAL FUNCTION INDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONEIN VALUE
LOGICAL DONEIN
COMMON /DONE/ DONEIN, DONECOMP
INDONE = DONEIN
END

LOGICAL FUNCTION COMPDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONECOMP VALUE
LOGICAL DONECOMP
COMMON /DONE/ DONEIN, DONECOMP
COMPDONE= DONECOMP
END

```

Notice that the gates are accessed through `COMMON` blocks. Each thread that calls this subroutine allocates a `thread_private` `WORK` array.

This subroutine contains a loop that tests `INDONE()`.

- The loop copies the input queue into the local `WORK` array, then does a significant amount of computational work that has been omitted for simplicity.

#### NOTE

The computational work is the main code that executes in parallel, if there is not a large amount of it, the overhead of setting up these parallel tasks and critical sections cannot be justified.

- The loop encompasses this computation, and also the section of code that copies the `WORK` array to the output queue.
- This construct allows final output to be written after all input has been used in computation.

Parallel synchronization  
Synchronizing code

- To avoid accessing the input queue while it is being filled or accessed by another thread, the section of code that copies it into the local `WORK` array is protected by a critical section.

**NOTE**

This section must be unconditionally locked as the computational threads cannot do something else until they receive their input.

Once the input queue has been copied, `THREAD_WRK` can perform its large section of computational code in parallel with whatever the other tasks are doing. After the computational section is finished, another unconditional critical section must be entered so that the results are written to the output queue. This prevents two threads from accessing the output queue at once.

Problems like this require performance testing and tuning to achieve optimal parallel efficiency. Variables such as the number of computational threads and the size of the I/O queues are adjusted to yield the best processor utilization.

This chapter discusses common optimization problems that occasionally occur when developing programs for SMP servers. Possible solutions to these problems are offered where applicable.

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 other cases, user error can cause similar problems in code that contains improperly used syntactically correct constructs or directives. If you encounter any of these problems, look for the following possible causes:

- Aliasing
- False cache line sharing
- Floating-point imprecision
- Invalid subscripts
- Misused directives and pragmas
- Triangular loops
- Compiler assumptions

**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

As described in the section “Inhibiting parallelization” on page 105, an alias is an alternate name for an object. Fortran `EQUIVALENCE` statements, C pointers, and procedure calls in both languages can potentially cause aliasing problems. Problems can and do occur at optimization levels `+O3` and above. However, code motion can also cause aliasing problems at optimization levels `+O1` and above.

Because they frequently use pointers, C programs are especially susceptible to aliasing problems. By default, the optimizer assumes that a pointer can point to any object in the entire application. Thus, any two pointers are potential aliases. The C compiler has two algorithms you can specify in place of the default: an ANSI-C aliasing algorithm and a type-safe algorithm.

The ANSI-C algorithm is enabled [disabled] through the `+O[no]ptrs_ansi` option.

The type-safe algorithm is enabled [disabled] by specifying the command-line option `+O[no]ptrs_strongly_typed`.

The defaults for these options are `+Onoptrs_ansi` and `+Onoptrs_strongly_typed`.

### ANSI algorithm

ANSI C provides strict type-checking. Pointers and variables cannot alias with pointers or variables of a different base type. The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant.

### Type-safe algorithm

The type-safe algorithm provides stricter type-checking. This allows the C compiler to use a stricter algorithm that eliminates many potential aliases found by the ANSI algorithm.

## Specifying aliasing modes

To specify an aliasing mode, use one of the following options on the C compiler command line:

- `+Optrs_ansi`
- `+Optrs_strongly_typed`

Additional C aliasing options are discussed in “Controlling optimization” on page 113.

## 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. (For information on Optimization Reports, see “Optimization Report” on page 151).

## Troubleshooting

### Aliasing

Avoid taking the address of any variable that is used as the iteration variable for a loop. To parallelize the loop in `subex`, use a temporary variable `i` as shown in the following code:

```
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 cannot find the loop's iteration count the compiler cannot 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. This is 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 is 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` is 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 this case, flow-sensitive analysis takes care of the aliasing. You can also 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, is altered on any iteration of the loop. The compiler cannot determine the stop value and cannot parallelize the loop.

## Troubleshooting

### Aliasing

```
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 is parallelized:

```
int n, ik[1000];
void foo()
{
    int i;

    for (i=0; i<n; i++)
        ik[i] = i;
}
```

---

## False cache line sharing

False cache line sharing is a form of cache thrashing. It occurs whenever two or more threads in a parallel program are assigning different data items in the same cache line. This section discusses how to avoid false cache line sharing by restructuring the data layout and controlling the distribution of loop iterations among threads.

Consider the following Fortran code:

```
REAL*4 A(8)
DO I = 1, 8
  A(I) = ...
  .
  .
  .
ENDDO
```

Assume there are eight threads, each executing one of the above iterations.  $A(1)$  is on a processor cache line boundary (32-byte boundary for V2250 servers) so that all eight elements are in the same cache line. Only one thread at a time can “own” the cache line, so not only is the above loop, in effect, run serially, but every assignment by a thread requires an invalidation of the line in the cache of its previous “owner.” These problems would likely eliminate any benefit of parallelization.

Taking all of the above into consideration, review the code:

```
REAL*4 B(100,100)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

Assume there are eight threads working on the  $I$  loop in parallel. The  $J$  loop cannot be parallelized because of the dependence. Table 60 on page 273 shows how the array maps to cache lines, assuming that  $B(1,1)$  is on a cache line boundary. Array entries that fall on cache line boundaries are in shaded cells. Array entries that fall on cache line boundaries are noted by hashmarks(#).

**Table 59** Initial mapping of array to cache lines

1, 1	1, 2	1, 3	1, 4	...	1, 99	1,100
2, 1	2, 2	2, 3	2, 4	...	2, 99	2,100
3, 1	3, 2	3, 3	3, 4	...	3, 99	3,100
4, 1	4, 2	4, 3	4, 4	...	4, 99	4,100
5, 1	5, 2	5, 3	5, 4	...	5, 99	5,100
6, 1	6, 2	6, 3	6, 4	...	6, 99	6,100
7, 1	7, 2	7, 3	7, 4	...	7, 99	7,100
8, 1	8, 2	8, 3	8, 4	...	8, 99	8,100
9, 1	9, 2	9, 3	9, 4	...	9, 99	9,100
10, 1	10, 2	10, 3	10, 4	...	10, 99	10,100
11, 1	11, 2	11, 3	11, 4	...	11, 99	11,100
12, 1	12, 2	12, 3	12, 4	...	12, 99	12,100
13, 1	13, 2	13, 3	13, 4	...	13, 99	13, 100
...	...	...	...	...	...	...
97, 1	97, 2	97, 3	97, 4	...	97, 99	97,100
98, 1	98, 2	98, 3	98, 4	...	98, 99	98,100
99, 1	99, 2	99, 3	99, 4	...	99, 99	99,100
100, 1	100, 2	100, 3	100, 4	...	100, 99	100,100

Array entries surrounded by hashmarks(#) are on cache line boundaries.

HP compilers, by default, give each thread about the same number of iterations, assigning (if necessary) one extra iteration to some threads. This happens until all iterations are assigned to a thread. Table 60 shows the default distribution of the  $\tau$  loop across 8 threads.

**Table 60**      **Default distribution of the  $\tau$  loop**

Thread ID	Iteration range	Number of iterations
0	1-12	12
1	13-25	13
2	26-37	12
3	38-50	13
4	51-62	12
5	63-75	13
6	76-87	12
7	88-100	13

This distribution of iterations causes threads to share cache lines. For example, thread 0 assigns the elements  $B(9:12, 1)$ , and thread 1 assigns elements  $B(13:16, 1)$  in the same cache line. In fact, every thread shares cache lines with at least one other thread. Most share cache lines with two other threads. This type of sharing is called false because it is a result of the data layout and the compiler's distribution of iterations. It is not inherent in the algorithm itself. Therefore, it is reduced or even removed by:

- 1 Restructuring the data layout by aligning data on cache line boundaries
- 2 Controlling the iteration distribution.

## Aligning data to avoid false sharing

Because false cache line sharing is partially due to the layout of the data, one step in avoiding it is to adjust the layout. Adjustments are typically made by aligning data on cache line boundaries. Aligning arrays generally improves performance. However, it can occasionally decrease performance.

The second step in avoiding false cache line sharing is to adjust the distribution of loop iterations. This is covered in “Distributing iterations on cache line boundaries” on page 275.

## Aligning arrays on cache line boundaries

Note the assumption that in the previous example, array `B` starts on a cache line boundary. The methods below force arrays in Fortran to start on cache line boundaries:

- Using uninitialized `COMMON` blocks (blocks with no `DATA` statements). These blocks start on 64-byte boundaries.
- Using `ALLOCATE` statements. These statements return addresses on 64-byte boundaries. This only applies to parallel executables.

The methods below force arrays in C to start on cache line boundaries:

- Using the functions `malloc` or `memory_class_malloc`. These functions return pointers on 64-byte boundaries.
- Using uninitialized global arrays or structs that are at least 32 bytes. Such arrays and structs are aligned on 64-byte boundaries.
- Using uninitialized data of the `external` storage class in C that is at least 32 bytes. Data is aligned on 64-byte boundaries.

## Distributing iterations on cache line boundaries

Recall that the default iteration distribution causes thread 0 to work on iterations 1-12 and thread 1 to work on iterations 13-25, and so on. Even though the cache lines are aligned across the columns of the array (see Table 60 on page 273), the iteration distribution still needs to be changed. Use the `CHUNK_SIZE` attribute to change the distribution:

```
REAL*4 B(112,100)
COMMON /ALIGNED/ B
C$DIR PREFER_PARALLEL (CHUNK_SIZE=16)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

You must specify a constant `CHUNK_SIZE` attribute. However, the ideal is to distribute work so that all but one thread works on the same number of whole cache lines, and the remaining thread works on any partial cache line. For example, given the following:

`NITS` = number of iterations

`NTHDS` = number of threads

`LSIZE` = line size in words (8 for 4-byte data, 4 for 8-byte data, 2 for 16-byte data) size in words (8 for 4-byte data)

the ideal `CHUNK_SIZE` would be:

$$\text{CHUNK\_SIZE} = \text{LSIZE} * (1 + ((1 + (\text{NITS} - 1) / \text{LSIZE}) - 1) / \text{NTHDS})$$

For the code above, these numbers are:

`NITS` = 100

`LSIZE` = 8 (aligns on V2250 boundaries for 4-byte data)

`NTHDS` = 8

```
CHUNK_SIZE = 8 * (1 + ((1 + (100 - 1) / 8) - 1) / 8)
            = 8 * (1 + ((1 + 12) - 1) / 8)
            = 8 * (1 + (12) / 8)
            = 8 * (1 + 1)
            = 16
```

`CHUNK_SIZE` = 16 causes threads 0, 1, ..., 6 to execute iterations 1-16, 17-32, ..., 81-96, respectively. Thread 7 executes iterations 97-100. As a result there is no false cache line sharing, and parallel performance is greatly improved.

## Troubleshooting

### False cache line sharing

You cannot specify the ideal `CHUNK_SIZE` for every loop. However, using `CHUNK_SIZE = x`

where  $x$  times the data size (in bytes) is an integral multiple of 32, eliminates false cache line sharing. This is only if the following two conditions below are met:

- The arrays are already properly aligned (as discussed earlier in this section).
- The first iteration accesses the first element of each array being assigned. For example, in a loop `DO I = 2, N`, because the loop starts at `I = 2`, the first iteration does not access the first element of the array. Consequently, the iteration distribution does not match the cache line alignment.

The number 32 is used because the cache line size is 32 bytes for V2250 servers.

## Thread-specific array elements

Sometimes a parallel loop has each thread update a unique element of a shared array, which is further processed by thread 0 outside the loop.

Consider the following Fortran code in which false sharing occurs:

```
REAL*4 S(8)
C$DIR LOOP_PARALLEL
DO I = 1, N
  .
  .
  .
  S(MY_THREAD()+1) = ... ! EACH THREAD ASSIGNS ONE ELEMENT OF S
  .
  .
  .
ENDDO
C$DIR NO_PARALLEL
DO J = 1, NUM_THREADS()
  = ...S(J) ! THREAD 0 POST-PROCESSES S
ENDDO
```

The problem here is that potentially all the elements of `S` are in a single cache line, so the assignments cause false sharing. One approach is to change the code to force the unique elements into different cache lines, as indicated in the following code:



```
REAL*4 S(8,8)
C$DIR LOOP_PARALLEL
DO I = 1, N
  .
  .
  .
  S(1,MY_THREAD()+1) = ... ! EACH THREAD ASSIGNS ONE ELEMENT OF S
  .
  .
  .
ENDDO
C$DIR NO_PARALLEL
DO J = 1, NUM_THREADS()
  = ...S(1,J) ! THREAD 0 POST-PROCESSES S
ENDDO
```

## Scalars sharing a cache line

Sometimes parallel tasks assign unique scalar variables that are in the same cache line, as in the following code:

```
COMMON /RESULTS/ SUM, PRODUCT
C$DIR BEGIN_TASKS
DO I = 1, N
  .
  .
  .
  SUM = SUM + ...
  .
  .
  .
ENDDO
C$DIR NEXT_TASK
DO J = 1, M
  .
  .
  .
  PRODUCT = PRODUCT * ...
  .
  .
  .
ENDDO
C$DIR END_TASKS
```

## Working with unaligned arrays

The most common cache-thrashing complication using arrays and loops occurs when arrays assigned within a loop are unaligned with each other. There are several possible causes for this:

- Arrays that are local to a routine are allocated on the stack.
- Array dummy arguments might be passed an element other than the first in the actual argument.
- Array elements might be assigned with different offset indexes.

Consider the following Fortran code:

```
COMMON /OKAY/ X(112,100)
      ...
CALL UNALIGNED (X(I,J))
      ...
SUBROUTINE UNALIGNED (Y)
REAL*4 Y(*)
      ! Y(1) PROBABLY NOT ON A CACHE LINE BOUNDARY
```

The address of  $Y(1)$  is unknown. However, if elements of  $Y$  are heavily assigned in this routine, it may be worthwhile to compute an alignment, given by the following formula:

$$\text{LREM} = \text{LSIZE} - \left( \left( \text{LOC}(Y(1)) - 4, \text{LSIZE} * x \right) + 4 \right) / x$$

where

$\text{LSIZE}$  is the appropriate cache line size in words

$x$  is the data size for elements of  $Y$

For this case,  $\text{LSIZE}$  on V2250 servers is 32 bytes in single precision words (8 words). Note that:

$$\left( \left( \text{MOD} \left( \text{LOC}(Y(1)) - 4, \text{LSIZE} * 4 \right) + 4 \right) / 4 \right)$$

returns a value in the set 1, 2, 3, ...,  $\text{LSIZE}$ , so  $\text{LREM}$  is in the range 0 to 7.

Then a loop such as:

```
DO I = 1, N
  Y(I) = ...
ENDDO
```

is transformed to:

```
C$DIR NO_PARALLEL
DO I = 1, MIN (LREM, N) ! 0 <= LREM < 8
  Y(I) = ...
ENDDO
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 16)
DO I = LREM+1, N
  ! Y(LREM+1) IS ON A CACHE LINE BOUNDARY
  Y(I) = ...
ENDDO
```

The first loop takes care of elements from the first (if any) partial cache line of data. The second loop begins on a cache line boundary, and is controlled with `CHUNK_SIZE` to avoid false sharing among the threads.

## Working with dependences

Data dependences in loops may prevent parallelization and prevent the elimination of false cache line sharing. If certain conditions are met, some performance gains are achieved.

For example, consider the following code:

```
COMMON /ALIGNED / P(128,128), Q(128,128), R(128,128)
REAL*4 P, Q, R
DO J = 2, 128
  DO I = 2, 127
    P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
    Q(I ,J) = SQRT (Q(I ,J-1) + 1./3.)
    R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
  ENDDO
ENDDO
```

Only the `I` loop is parallelized, due to the loop-carried dependences in the `J` loop. It is impossible to distribute the iterations so that there is no false cache line sharing in the above loop. If all loops that refer to these arrays always use the same offsets (which is unlikely) then you could make dimension adjustments that would allow a better iteration distribution.

For example, the following would work well for 8 threads:

```
COMMON /ADJUSTED/ P(128,128), PAD1(15), Q(128,128),
> PAD2(15), R(128,128)
DO J = 2, 128
C$DIR PREFER_PARALLEL (CHUNK_SIZE=16)
  DO I = 2, 127
    P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
    Q(I ,J) = SQRT (Q(I ,J-1) + 1./3.)
    R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
  ENDDO
ENDDO
```

## Troubleshooting

### False cache line sharing

Padding 60 bytes before the declarations of both `Q` and `R` causes the `P(1, J)`, `Q(2, J)`, and `R(3, J)` to be aligned on 64-byte boundaries for all `J`. Combined with a `CHUNK_SIZE` of 16, this causes threads to assign data to unique whole cache lines.

You can usually find a mix of all the above problems in some CPU-intensive loops. You cannot avoid all false cache line sharing, but by careful inspection of the problems and careful application of some of the workarounds shown here, you can significantly enhance the performance of your parallel loops.

## 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 is 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 differs 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 (reductions) 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
```

## Troubleshooting

### Floating-point imprecision

```
C$DIR NEXT_TASK
      DO I = 1, 10000
        X = X + C(I) + D(I)
      ENDDO
C$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 run in. This means that the roundoff error accumulates 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.

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 and C++, or `REAL*8` rather than `REAL*4` in Fortran). Testing floating-point numbers for exact equality is strongly discouraged.

## Enabling sudden underflow

By default, PA-RISC processor hardware represents a floating point number in denormalized format when the number is tiny. A floating point number is considered tiny if its exponent field is zero but its mantissa is nonzero. This practice is extremely costly in terms of execution time and seldom provides any benefit.

You can enable sudden underflow (flush to zero) of denormalized values by passing the `+FPD` flag to the linker. This is done using the `-W` compiler option.

For more information, refer to the *HP-UX Floating-Point Guide*.

The following example shows an `f90` command line issuing this command:

```
%f90 -w1,+FPD prog.f
```

This command line compiles the program `prog.f` and instructs the linker to enable sudden underflow.

---

## 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 result in a core dump.

Use the command-line option `-C` (check subscripts) with `f90` to check that each subscript is within its array bounds. See the `f90(1)` man page for more information. The C and aC++ compilers do not have an option corresponding to the Fortran compiler's `-C` option.

## Misused directives and pragmas

Misused directives and pragmas are a common cause of wrong answers. Some of the more common misuses of directives and pragmas involve the following:

- Loop-carried dependences
- Reductions
- Nondeterminism of parallel execution

Descriptions of and methods for avoiding the items listed above are described in the sections below.

### Loop-carried dependences

Forcing parallelization of a loop containing a call is safe only if the called routine contains no dependences.

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 dependence (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 section “Conditions that inhibit data localization” on page 59.

You cannot safely parallelize a loop that contains any kind of LCD, except by using ordered sections around the LCDs as described in the section “Ordered sections” on page 248. Also see the section “Inhibiting parallelization” on page 105.



The MAIN section of the Fortran program below initializes A, calls CALC, and outputs the new array values. In subroutine CALC, the indirect index used in A(IN(I)) introduces a potential dependence 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
```

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, when compiling with +O3 +Oparallel.

```
          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
```

## Reductions

Reductions are a special class of dependence 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 dependence between the references to `A(I)` on line 6 and the assignment to `A(JA(J))` on line 7. The compiler does not realize that the values of the elements of `JA` never coincide with the values of `I`. Assuming that they might collide, the compiler conservatively avoids parallelizing the loop.

```
DO I = 1,100
  JA(I) = I + 10
ENDDO
DO I = 1, 100
  DO J = I, 100
    A(I) = A(I) + B(J) * C(J)      !LINE 6
    A(JA(J)) = B(J) + C(J)       !LINE 7
  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 to 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 6. The apparent dependence on line 7 is properly handled because of the directive. The resulting code runs fast but produces incorrect answers.

To solve this problem, distribute the  $J$  loop, isolating the reduction from the other statements, as shown in the following Fortran example:

```
DO I = 1, 100
  DO J = I, 100
    A(I) = A(I) + B(J) * C(J)
  ENDDO
ENDDO
C$DIR NO_LOOP_DEPENDENCE(A)
DO I = 1, 100
  DO J = I, 100
    A(JA(J)) = B(J) + C(J)
  ENDDO
ENDDO
```

The apparent dependence is removed, and both loops are optimized.

## 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 dependence exists, the results are unpredictable and can vary from one execution to the next.

Consider the following Fortran code:

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

The compiler does not parallelize this code as written because of the dependence on  $A(I)$ . This dependence requires that the original value of  $A(I+1)$  be 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.

Loops containing dependences can sometimes be manually parallelized using the `LOOP_PARALLEL(ORDERED)` directive as described in “Parallel synchronization” on page 233. Unless you are sure that no loop-carried dependence exists, it is safest to let the compiler choose which loops to parallelize.

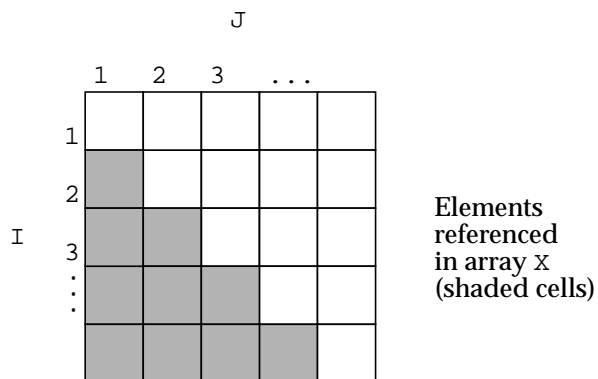
---

## Triangular loops

A triangular loop is a loop nest with an inner loop whose upper or lower bound (but not both) is a function of the outer loop's index. Examples of a lower triangular loop and an upper triangular loop are given below. To simplify explanations, only Fortran examples are provided in this section.

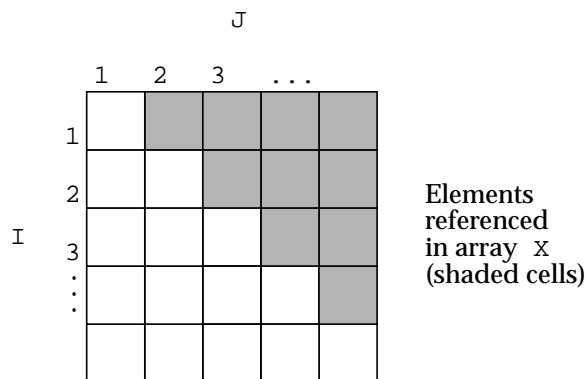
### Lower triangular loop

```
DO J = 1, N
  DO I = J+1, N
    F(I) = F(I) + ... + X(I,J) + ...
```



### Upper triangular loop

```
DO J = 1, N
  DO I = 1, J-1
    F(I) = F(I) + ... + X(I,J) + ...
```



While the compiler can usually auto-parallelize one of the outer or inner loops, there are typically performance problems in either case:

- If the outer loop is parallelized by assigning contiguous chunks of iterations to each of the threads, the load is severely unbalanced. For example, in the lower triangular example above, the thread doing the last chunk of iterations does far less work than the thread doing the first chunk.
- If the inner loop is auto-parallelized, then on each outer iteration in the  $J$  loop, the threads are assigned to work on a different set of iterations in the  $I$  loop, thus losing access to some of their previously encached elements of  $F$  and thrashing each other's caches in the process.

By manually controlling the parallelization, you can greatly improve the performance of a triangular loop. Parallelizing the outer loop is generally more beneficial than parallelizing the inner loop. The next two sections explain how to achieve the enhanced performance.

## Parallelizing the outer loop

Certain directives allow you to control the parallelization of the outer loop in a triangular loop to optimize the performance of the loop nest.

For the outer loop, assign iterations to threads in a balanced manner. The simplest method is to assign the threads one at a time using the `CHUNK_SIZE` attribute:

```
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 1)
      DO J = 1, N
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

This causes each thread to execute in the following manner:

```
      DO J = MY_THREAD() + 1, N, NUM_THREADS()
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

where  $0 \leq \text{MY\_THREAD}() < \text{NUM\_THREADS}()$

In this case, the first thread still does more work than the last, but the imbalance is greatly reduced. For example, assume  $N = 128$  and there are 8 threads. Then the default parallel compilation would cause thread 0 to do  $J = 1$  to 16, resulting in 1912 inner iterations, whereas thread 7 does  $J = 113$  to 128, resulting in 120 inner iterations. With `chunk_size = 1`, thread 0 does 1072 inner iterations, and thread 7 does 1023.

## Parallelizing the inner loop

If the outer loop cannot be parallelized, it is recommended that you parallelize the inner loop if possible. There are two issues to be aware of when parallelizing the inner loop:

- Cache thrashing

Consider the parallelization of the following inner loop:

```
      DO J = I+1, N
        F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
```

where  $I$  varies in the outer loop iteration.

The default iteration distribution has each thread processing a contiguous chunk of iterations of approximately the same number as every other thread. The amount of work per thread is about the same; however, from one outer iteration to the next, threads work on different elements in *F*, resulting in cache thrashing.

- The overhead of parallelization

If the loop cannot be interchanged to be outermost (or at least outermore), then the overhead of parallelization is compounded by the number of outer loop iterations.

The scheme below assigns “ownership” of elements to threads on a cache line basis so that threads always work on the same cache lines and retain data locality from one iteration to the next. In addition, the `parallel` directive is used to spawn threads just once. The outer, nonparallel loop is replicated on all processors, and the inner loop iterations are manually distributed to the threads.

```

C F IS KNOWN TO BEGIN ON A CACHE LINE BOUNDARY
  NTHD = NUM_THREADS()
  CHUNK = 8                ! CHUNK * DATA SIZE (4 BYTES)
                           ! EQUALS PROCESSOR CACHE LINE SIZE;
                           ! A SINGLE THREAD WORKS ON CHUNK = 8
                           ! ITERATIONS AT A TIME
  NTCHUNK = NTHD * CHUNK  ! A CHUNK TO BE SPLIT AMONG THE THREADS
  ...
C$DIR PARALLEL, PARALLEL_PRIVATE(ID, JS, JJ, J, I)
  ID = MY_THREAD() + 1    ! UNIQUE THREAD ID
  DO I = 1, N
    JS = ((I+1 + NTCHUNK-1 - ID*CHUNK) / NTCHUNK) * NTCHUNK
    >   + (ID-1) * CHUNK + 1
    DO JJ = JS, N, NTCHUNK
      DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
        F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
      ENDDO
    ENDDO
  ENDDO
C$DIR END_PARALLEL

```

The idea is to assign a fixed ownership of cache lines of *F* and to assign a distribution of those cache lines to threads that keeps as many threads busy computing whole cache lines for as long as possible. Using `CHUNK = 8` for 4-byte data makes each thread work on 8 iterations covering a total of 32 bytes—the processor cache line size for V2250 servers.

## Troubleshooting

### Triangular loops

In general, set `CHUNK` equal to the smallest value that multiplies by the data size to give a multiple of 32 (the processor cache line size on V2250 servers). Smaller values of `CHUNK` keep most threads busy most of the time.

Because of the ever-decreasing work in the triangular loop, there are fewer cache lines left to compute than there are threads. Consequently, threads drop out until there is only one thread left to compute those iterations associated with the last cache line. Compare this distribution to the default distribution that causes false cache line sharing and consequent thrashing when all threads attempt to compute data into a few cache lines. See “False cache line sharing” on page 271 in this chapter.

The scheme above maps a sequence of `NTCHUNK`-sized blocks over the `F` array. Within each block, each thread owns a specific cache line of data. The relationship between data, threads, and blocks of size `NTCHUNK` is shown in Figure 19 on page 293.



**Figure 19**      **Data ownership by CHUNK and NTCHUNK blocks**

**NTCHUNK 1**

<b>CHUNKs of F</b>	<b>Associated thread</b>
F(1) ... F(8)	thread 0
F(9) ... F(16)	thread 1
F(17) ... F(24)	thread 2
F(33) ... F(40)	thread 3
F(41) ... F(48)	thread 4
F(25) ... F(32)	thread 5
F(49) ... F(56)	thread 6
F(57) ... F(64)	thread 7

**NTCHUNK 2**

<b>CHUNKs of F</b>	<b>Associated thread</b>
F(65) ... F(72)	thread 0
F(73) ... F(80)	thread 1
F(81) ...	...

CHUNK is the number of iterations a thread works on at one time. The idea is to make a thread work on the same elements of F from one iteration of I to the next (except for those that are already complete).

The scheme above causes thread 0 to do all work associated with the cache lines starting at F(1), F(1+NTCHUNK), F(1+2\*NTCHUNK), and so on. Likewise, thread 1 does the work associated with the cache lines starting at F(9), F(9+NTCHUNK), F(9+2\*NTCHUNK), and so on.

Troubleshooting  
Triangular loops

If a thread assigns certain elements of  $F$  for  $I = 2$ , then it is certain that the same thread encached those elements of  $F$  in iteration  $I = 1$ . This eliminates cache thrashing among the threads.

## Examining the code

Having established the idea of assigning cache line ownership, consider the following Fortran code in more detail:

```
C$DIR PARALLEL, PARALLEL_PRIVATE( ID, JS, JJ, J, I)
  ID = MY_THREAD() + 1    ! UNIQUE THREAD ID
  DO I = 1, N
    JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
    >   + (ID-1) * CHUNK + 1
    DO JJ = JS, N, NTCHUNK
      DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
        F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
      ENDDO
    ENDDO
  ENDDO
C$DIR END_PARALLEL
```

```
C$DIR PARALLEL, PARALLEL_PRIVATE( ID, JS, JJ, J, I)
```

Spawns threads, each of which begins executing the statements in the parallel region. Each thread has a private version of the variables  $ID$ ,  $JS$ ,  $JJ$ ,  $J$ , and  $I$ .

```
ID = MY_THREAD() + 1    ! UNIQUE THREAD ID
```

Establishes a unique  $ID$  for each thread, in the range 1 to `num_threads()`.

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

Executes all threads of the  $I$  loop redundantly (instead of thread 0 executing it alone).

```
JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
    + (ID-1) * CHUNK + 1
```

Determines, for a given value of  $I+1$ , which  $NTCHUNK$  the value  $I+1$  falls then. Then it assigns a unique  $CHUNK$  of it to each thread  $ID$ . Suppose that there are  $ntc$   $NTCHUNK$ s, where  $ntc$  is approximately  $N/NTCHUNK$ . Then the expression:

$(I+1 + NTCHUNK-1 - ID*CHUNK) / NTCHUNK$

returns a value in the range 1 to *ntc* for a given value of *I+1*. Then the expression:

$((I+1 + NTCHUNK-1 - ID*CHUNK) / NTCHUNK) * NTCHUNK$

identifies the start of an NTCHUNK that contains *I+1* or is immediately above *I+1* for a given value of *ID*.

For the NTCHUNK that contains *I+1*, if the cache lines owned by a thread either contain *I+1* or are above *I+1* in memory, this expression returns this NTCHUNK. If the cache lines owned by a thread are below *I+1* in this NTCHUNK, this expression returns the next highest NTCHUNK. In other words, if there is no work for a particular thread to do in this NTCHUNK, then start working in the next one.

$(ID-1) * CHUNK + 1$

identifies the start of the particular cache line for the thread to compute within this NTCHUNK.

DO JJ = JS, N, NTCHUNK

runs a unique set of cache lines starting at its specific JS and continuing into succeeding NTCHUNKS until all the work is done.

DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)

performs the work within a single cache line. If the starting index (*I+1*) is greater than the first element in the cache line (*JS*) then start with *I+1*. If the ending index (*N*) is less than the last element in the cache line, then finish with *N*.

The following are observations of the preceding loops:

- Most of the “complicated” arithmetic is an outer loop iterations.
- You can replace divides with shift instructions because they involve powers of two.
- If this application were to be run on an V2250 single-node machine, it would be appropriate to choose a chunk size of 8 for 4-byte data.

## Compiler assumptions

Compiler assumptions can produce faulty optimized code when the source code contains:

- Iterations by zero
- Trip counts that may overflow at optimization levels +O2 and above

Descriptions of, and methods for, avoiding the items listed above are in the following sections.

### Incrementing by zero

The compiler assumes that whenever a variable is being incremented on each iteration of a loop, the variable is being incremented by a loop-invariant amount other than 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 symptoms described below occur.

The following Fortran code shows two loops that increment by zero:

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)
J = 1
DO I = 1, 100, IZR ! INCREMENT VALUE OF 0 IS
                  ! NON-STANDARD
    A(I) = B(I)
ENDDO
PRINT *, A(11)
DO I = 1, 100
    J = J + IZR
    B(I) = A(J)
    A(J) = C(I)
ENDDO
PRINT *, A(1)
PRINT *, B(11)
END
```

Because `IZR` is an argument passed to `SUB1`, the compiler does not detect that `IZR` has been set to zero. Both loops parallelize at `+O3 +Oparallel +Onodynse1`.

The loops compile at `+O3`, but the first loop, which specifies the step as part of the `DO` statement (or as part of the `for` statement in C), attempts to parcel out loop iterations by a step of `IZR`. At runtime, this loop is infinite.

Due to dependences, the second loop would not behave predictably when parallelized—if it were ever reached at runtime. The compiler does not detect the dependences because it assumes `J` is an induction variable.

## Trip counts that may overflow

Some loop optimizations at `+O2` and above may cause the variable on which the trip count is based to overflow. A loop's trip count is the number of times the loop executes. The compiler assumes that each induction variable is increasing (or decreasing) without overflow during the loop. Any overflowing induction variable may be used by the compiler as a basis for the trip count. The following sections discuss when this overflow may occur and how to avoid it.

## Linear test replacement

When optimizing loops, the compiler often disregards the original induction variable, using instead a variable or value that better indicates the actual stride of the loop. A loop's stride is the value by which the iteration variable increases on each iteration. By picking the largest possible stride, the compiler reduces the execution time of the loop by reducing the number of arithmetic operations within each iteration.

The Fortran code below contains an example of a loop in which the induction variable may be replaced by the compiler:

```

ICONST = 64
ITOT = 0
DO IND = 1,N
  IPACK = (IND*1024)*ICONST**2
  IF(IPACK .LE. (N/2)*1024*ICONST**2)
>    ITOT = ITOT + IPACK
  .
  .
  .
ENDDO
END

```

Troubleshooting  
Compiler assumptions

Executing this loop using `IND` as the induction variable with a stride of 1 would be extremely inefficient. Therefore, the compiler picks `IPACK` as the induction variable and uses the amount by which it increases on each iteration,  $1024 \cdot 64^2$  or  $2^{22}$ , as the stride.

The *trip count* ( $N$  in the example), or just *trip*, is the number of times the loop executes, and the *start* value is the initial value of the induction variable.

Linear test replacement, a standard optimization at levels `+O2` and above, normally does not cause problems. However, when the loop stride is very large a large trip count can cause the loop limit value ( $start + ((trip - 1) \cdot stride)$ ) to overflow.

In the code above, the induction variable is a 4-byte integer, which occupies 32 bits in memory. That means if  $start + ((trip - 1) \cdot stride)$  ( $1 + ((N - 1) \cdot 2^{22})$ ) is greater than  $2^{31} - 1$ , the value overflows into the sign bit and is treated as a negative number. If the stride value is negative, the absolute value of  $start + ((trip - 1) \cdot stride)$  must be not exceed  $2^{31}$ . When a loop has a positive stride and the trip count overflows, the loop stops executing when the overflow occurs because the limit becomes negative—assuming a positive stride—and the termination test fails.

Because the largest allowable value for  $start + ((trip - 1) \cdot stride)$  is  $2^{31} - 1$ , the start value is 1, and the stride is  $2^{22}$ , the maximum trip count for the loop is found.

The stride, trip, and start values for a loop must satisfy the following inequality:

$$start + ((trip - 1) * stride) \leq 2^{31}$$

The start value is 1, so trip is solved as follows:

$$start + ((trip - 1) * stride) \leq 2^{31}$$

$$1 + (trip - 1) * 2^{22} \leq 2^{31}$$

$$(trip - 1) * 2^{22} \leq 2^{31} - 1$$

$$trip - 1 \leq 2^9 - 2^{-22}$$

$$trip \leq 2^9 - 2^{-22} + 1$$

$$trip \leq 512$$

The maximum value for  $n$  in the given loop, then, is 512.

**NOTE**

If you find that certain loops give wrong answers at optimization levels +O2 or higher, the problem may be test replacement. If you still want to optimize these loops at +O2 or above, restructure them to force the compiler to choose a different induction variable.

**Large trip counts at +O2 and above**

When a loop is optimized at level +O2 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). Loops with trip counts that cannot be determined at compile time but that exceed  $2^{31} - 1$  at runtime yield wrong answers.

This limitation only applies at optimization levels +O2 and above.

A loop with a trip count that overflows 32 bits is optimized by manually strip mining the loop.

Troubleshooting  
Compiler assumptions



---

# A

---

## Porting CPSlib functions to pthreads

---

### Introduction

The Compiler Parallel Support Library (CPSlib) is a library of thread management and synchronization routines that was initially developed to control parallelism on HP's legacy multinode systems. Most programs fully exploited their parallelism using higher-level devices such as automatic parallelization, compiler directives, and message-passing. CPSlib, however, provides a lower-level interface for the few cases that required it.

With the introduction of the V2250 series server, HP recommends the use of POSIX threads (pthreads) for purposes of thread management and parallelism. Pthreads provide portability for programmers who want to use their applications on multiple platforms.

This appendix describes how CPSlib functions map to pthread functions, and how to write a pthread program to perform the same tasks as CPSlib functions. Topics included in this chapter include:

- Accessing pthreads
- Symmetric parallelism
- Asymmetric parallelism
- Synchronization using high-level functions
- Synchronization using low-level functions

If you are running on a server released prior to the V2250 and require explicit information on CPSlib, refer to the *Exemplar Programming Guide for HP-UX systems*.

## Accessing pthreads

When you use pthreads routines, your program must include the `<pthread.h>` header file and the pthreads library must be explicitly linked to your program.

For example, assume the program `prog.c` contains calls to pthreads routines. To compile the program so that it links in the pthreads library, issue the following command:

```
% cc -D_POSIX_C_SOURCE=199506L prog.c -lpthread
```

The `-D_POSIX_C_SOURCE=199506L` string indicates the appropriate POSIX revision level. In the example above, the level is indicated as 199506L.

## Mapping CPSlib functions to pthreads

Table 61 shows the mapping of the CPSlib functions to pthread functions. Where applicable, a pthread function is listed as corresponding to the appropriate CPSlib function. For instances where there is no corresponding pthread function, pthread examples that mimic CPSlib functionality are provided.

The CPSlib functions are grouped by type: barriers, informational, low-level locks, low-level counter semaphores, symmetric and asymmetric, and mutexes.

**Table 61** CPSlib library functions to pthreads mapping

CPSlib function	Maps to pthread function
Symmetric parallel functions	
cps_nsthreads	N/A See “Symmetric parallelism” on page 310 for more information.
cps_ppcall	N/A See “Symmetric parallelism” on page 310 for more information. Nesting is not supported in this example.
cps_ppcalln	N/A See “Symmetric parallelism” on page 310 for more information.
cps_ppcallv	N/A No example provided.
cps_stid	N/A See “Symmetric parallelism” on page 310 for more information.

Porting CPSlib functions to pthreads  
 Mapping CPSlib functions to pthreads

<b>CPSlib function</b>	<b>Maps to pthread function</b>
cps_wait_attr	N/A  See “Symmetric parallelism” on page 310 for more information.
<b>Asymmetric parallel functions</b>	
cps_thread_create	pthread_create  See “Asymmetric parallelism” on page 321 for more information.
cps_thread_createn	pthread_create  Only supports passing of one argument.  See “Asymmetric parallelism” on page 321 for more information.
cps_thread_exit	pthread_exit  See “Asymmetric parallelism” on page 321 for more information.
cps_thread_register_lock	This function was formerly used in conjunction with m_lock. It is now obsolete, and is replaced with one call to pthread_join.  See “Asymmetric parallelism” on page 321 for more information.
cps_thread_wait	N/A  No example available.
<b>Informational</b>	
cps_complex_cpus	pthread_num_processors_np  The HP pthread_num_processors_np function returns the number of processors on the machine.

<b>CPSlib function</b>	<b>Maps to pthread function</b>
<code>cps_complex_nodes</code>	N/A  This functionality can be added using the appropriate calls in your <code>ppcall</code> code.
<code>cps_complex_nthreads</code>	N/A  This functionality can be added using the appropriate calls in your <code>ppcall</code> code.
<code>cps_is_parallel</code>	N/A  See the <code>ppcall.c</code> example on page 310 for more information.
<code>cps_plevel</code>	Because pthreads have no concept of levels, this function is obsolete.
<code>cps_set_threads</code>	N/A  See the <code>ppcall.c</code> example on page 310 for more information.
<code>cps_topology</code>	Use <code>pthread_num_processors_np()</code> to set up your configuration as a single-node machine.
<b>Synchronization using high-level barriers</b>	
<code>cps_barrier</code>	N/A  See the <code>my_barrier.c</code> example in on page 324 for more information.
<code>cps_barrier_alloc</code>	N/A  See the <code>my_barrier.c</code> example in on page 324 for more information.

Porting CPSlib functions to pthreads  
 Mapping CPSlib functions to pthreads

<b>CPSlib function</b>	<b>Maps to pthread function</b>
cps_barrier_free	N/A  See the <code>my_barrier.c</code> example in on page 324 for more information.
<b>Synchronization using high-level mutexes</b>	
cps_limited_spin_mutex_alloc	pthread_mutex_init  The CPS mutex allocate functions allocated memory and initialized the mutex. When you use pthread mutexes, you must use <code>pthread_mutex_init</code> to allocate the memory and initialize it.  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.
cps_mutex_alloc	pthread_mutex_init  The CPS mutex allocate functions allocated memory and initialized the mutex. When you use pthread mutexes, you must use <code>pthread_mutex_init</code> to allocate the memory and initialize it.  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.
cps_mutex_free	pthread_mutex_destroy  <code>cps_mutex_free</code> formerly uninitialed the mutex, and called <code>free</code> to release memory. When using pthread mutexes, you must first call <code>pthread_mutex_destroy</code> .  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.
cps_mutex_lock	pthread_mutex_lock  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.

<b>CPSlib function</b>	<b>Maps to pthread function</b>
<code>cps_mutex_trylock</code>	<code>pthread_mutex_trylock</code>  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.
<code>cps_mutex_unlock</code>	<code>pthread_mutex_unlock</code>  See <code>pth_mutex.c</code> on page 324 for a description of using pthreads.
<b>Synchronization using low-level locks</b>	
<code>[mc]_cond_lock</code>	<code>pthread_mutex_trylock</code>
<code>[mc]_free32</code>	<code>pthread_mutex_destroy</code>  <code>cps_mutex_free</code> formerly uninitialized the mutex, and called free to release memory. When using pthread mutexes, you must call <code>pthread_mutex_destroy</code> .
<code>[mc]_init32</code>	<code>pthread_mutex_init</code>
<code>[mc]_lock</code>	<code>pthread_mutex_lock</code>
<code>[mc]_unlock</code>	<code>pthread_mutex_unlock</code>
<b>Synchronization using low-level counter semaphores</b>	
<code>[mc]_fetch32</code>	N/A  See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.
<code>[mc]_fetch_and_add32</code>	N/A  See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.
<code>[mc]_fetch_and_clear32</code>	N/A  See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.

Porting CPSlib functions to pthreads  
Mapping CPSlib functions to pthreads

<b>CPSlib function</b>	<b>Maps to pthread function</b>
<code>[mc]_fetch_and_dec32</code>	N/A See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.
<code>[mc]_fetch_and_inc32</code>	N/A See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.
<code>[mc]_fetch_and_set32</code>	N/A See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.
<code>[mc]_init32</code>	N/A See <code>fetch_and_inc.c</code> example on page 329 for a description of using pthreads.



## Environment variables

Unlike CPSlib, pthreads does not use environment variables to establish thread attributes. pthreads implements function calls to achieve the same results. However, when using the HP compiler set, the environment variables below must be set to define attributes.

The table below describes the environment variables and how pthreads handles the same or similar tasks.

The environment variables below must be set for use with the HP compilers if you are not explicitly using pthreads.

**Table 62** CPSlib environment variables

Environment variable	Description	How handled by pthreads
MP_NUMBER_OF_THREADS	Sets the number of threads that the compiler allocates at startup time.	By default, under HP-UX you can create more threads than you have processors for.
MP_IDLE_THREADS_WAIT	Indicates how idle compiler threads should wait.	The values can be: -1 - spin wait; 0 - suspend wait; N - spin suspend where $N > 0$ .
CPS_STACK_SIZE	Tells the compiler what size stack to allocate for all it's child threads. The default stacksize is 80 Mbyte.	Pthreads allow you to set the stack size using attributes. The attribute call is <code>pthread_attr_setstacksize</code> . The value of CPS_STACK_SIZE is specified in Kbytes.

## Using pthreads

Some CPSlib functions map directly to existing pthread functions, as shown in Table 61 on page 303. However, certain CPSlib functions, such as `cps_plevel`, are obsolete in the scope of pthreads. While about half of the CPSlib functions do not map to pthreads, their tasks can be simulated by the programmer.

The examples presented in the following sections demonstrate various constructs that can be programmed to mimic unmappable CPSlib functions in pthreads. The examples shown here are provided as a first step in replacing previous functionality provided by CPSlib with POSIX thread standard calls.

This is not a tutorial in pthreads, nor do these examples describe complex pthreads operations, such as nesting. For a definitive description of how to use pthreads functions, see the book *Threadtime* by Scott Norton and Mark D. Dipasquale.

## Symmetric parallelism

Symmetric parallel threads are spawned in CPSlib using `cps_ppcall()` or `cps_ppcalln()`. There is no logical mapping of these CPSlib functions to pthread functions. However you can create a program, similar to the one shown in the `ppcall.c` example below, to achieve the same results.

This example also includes the following CPSlib thread information functions:

- `my_nsthreads` (a map created for `cps_nsthreads`) returns the number of threads in the current spawn context.
- `my_stid` (a map created for `cps_stid`) returns the spawn thread ID of the calling thread.

The `ppcall.c` example performs other tasks associated with symmetrical thread processing, including the following:

- Allocates a cell barrier data structure based upon the number of threads in the current process by calling `my_barrier_alloc`

- Provides a barrier for threads to “join” or synchronize after parallel work is completed by calling `my_join_barrier`
- Creates data structures for threads created using `pthread_create`
- Uses the `CPS_STACK_SIZE` environment variable to determine the stacksize
- Determines the number of threads to create by calling `pthread_num_processors_np()`
- Returns the number of threads by calling `my_nsthreads()`
- Returns the `is_parallel` flag by calling `my_is_parallel()`

### **ppcall.c**

```
/*
 * ppcall.c
 * function
 * Symmetric parallel interface to using pthreads
 * called my_thread package.
 */

#ifdef _HPUX_SOURCE
#define _HPUX_SOURCE
#endif

#include <spp_prog_model.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
#include "my_ppcall.h"

#define K      1024
#define MB    K*K

struct thread_data {
    int      stid;
    int      nsthreads;
    int      release_flag; r};
};

typedef struct thread_data thread_t;
typedef struct thread_data *thread_p;

#define WAIT_UNKNOWN0
#define WAIT_SPIN1
#define WAIT_SUSPEND2

#define MAX_THREADS64

#define W_CACHE_SIZE  8
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
#define B_CACHE_SIZE    32

typedef struct {
    int volatile    c_cell;
    int             c_pad[W_CACHE_SIZE-1];
} cell_t;

#define ICELL_SZ        (sizeof(int)*3+sizeof(char *))

struct cell_barrier {
    int             br_c_magic;
    int volatile    br_c_release;
    char *          br_c_free_ptr;
    int             br_c_cell_cnt;
    char            br_c_pad[B_CACHE_SIZE-ICELL_SZ];
    cell_t          br_c_cells[1];
};

#define BR_CELL_T_SIZE(x) (sizeof(struct cell_barrier) +
    (sizeof(cell_t)*x))

/*
 * ALIGN - to align objects on specific alignments (usually on
 * cache line boundaries.
 *
 * arguments
 *     obj- pointer object to align
 *     alignment- alignment to align obj on
 *
 * Notes:
 *     We cast obj to a long, so that this code will work in
 *     either narrow or wide modes of the compilers.
 */
#define ALIGN(obj, alignment)\
    (((long) obj) + alignment - 1) & ~(alignment - 1))

typedef struct cell_barrier * cell_barrier_t;

/*
 * File Variable Dictionary:
 *
 * my_thread_mutex- mutex to control access to the following:
 * my_func, idle_release_flag, my_arg,
 * my_call_thread_max, my_threads_are_init,
 * my_threads_are_parallel.
 *
 * idle_release_flag    - flag to release spinning
 *                       idle threads
 * my_func               - user specified function to call
 * my_arg                - argument to pass to my_func
 * my_call_thread_max    - maximum number of threads
 *                       needed on this ppcall
 * my_threads_are_init   - my thread package init flag
 * my_threads_are_parallel - we are executing parallel
 *                       code flag
 * my_thread_ids         - list of child thread ids
 */
```

```
*      my_barrier          - barrier used by the join
*      my_thread_ptr      - the current thread thread
*                          - pointer in thread-private
*                          memory.
*/

static pthread_mutex_t my_thread_mutex =
PTHREAD_MUTEX_INITIALIZER;
static int volatile    idle_release_flag = 0;
static void           (*my_func)(void *);
static void           *my_arg;
static int            my_call_thread_max;
static int            my_stacksize = 8*MB;
static int            thread_count = 1;
static int            my_threads_are_init = 0;
static int volatile   my_threads_are_parallel = 0;
static pthread_t      my_thread_ids[MAX_THREADS];
static cell_barrier_t  my_barrier;

static thread_p thread_private my_thread_ptr;

/*
 * my_barrier_alloc
 *   Allocate cell barrier data structure based upon the
 *   number of threads that are in the current process.
 *
 * arguments
 *   brc - pointer pointer to the user cell barrier
 *   n   - number of threads that will use this barrier
 *
 * return
 *   0- success
 *  -1- failed to allocate cell barrier
 */

static int
my_barrier_alloc(cell_barrier_t *brc, int n)
{
    cell_barrier_t b;
    char *p;
    int i;

    /*
     * Allocate cell barrier for 'n' threads
     */
    if ( (p = (char *) malloc(BR_CELL_T_SIZE(n))) == 0 )
        return -1;

    /*
     * Align the barrier on a cache line for maximum
     * performance.
     */
}
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
b = (cell_barrier_t) ALIGN(p, B_CACHE_SIZE);
b->br_c_magic = 0x4200beef;
b->br_c_cell_cnt = n; /* keep track of the # of threads */
b->br_c_release = 0; /* initialize release flag */
b->br_c_free_ptr = p; /* keep track of original malloc ptr */

for(i = 0; i < n; i++ )
    b->br_c_cells[i].c_cell = 0; /* zero the cell flags */

*brc = b;

return 0;
}
/*
 * my_join_barrier
 * Provide a barrier for all threads to sync up at, after
 * they have finished performing parallel work.
 *
 * arguments
 * b - pointer to cell barrier
 * id - id of the thread (need to be in the
 * range of 0 - (N-1), where N is the
 * number of threads).
 *
 * return
 * none
 */

static void
my_join_barrier(cell_barrier_t b, int id)
{
    int i, key;

    /*
     * Get the release flag value, before we signal that we
     * are at the barrier.
     */
    key = b->br_c_release;

    if ( id == 0 ) {
        /*
         * make thread 0 (i.e. parent thread) wait for the child
         * threads to show up.
         */
        for( i = 1; i < thread_count; i++ ) {
            /*
             * wait on the Nth cell
             */
            while ( b->br_c_cells[i].c_cell == 0 )
                /* spin */;

            /*
             * We can reset the Nth cell now,
             * because it is not being used anymore
             * until the next barrier.
             */
        }
    }
}
```

```
b->br_c_cells[i].c_cell = 0;
}

/*
 * signal all of the child threads to leave the barrier.
 */
++b->br_c_release;
} else {
/*
 * signal that the Nth thread has arrived at the barrier.
 */
b->br_c_cells[id].c_cell = -1;

while ( key == b->br_c_release )
/* spin */;
}

/*
 * idle_threads
 * All of the process child threads will execute this
 * code. It is the idle loop where the child threads wait
 * for parallel work.
 * arguments
 * thr- thread pointer
 *
 * algorithm:
 * Initialize some thread specific data structures.
 * Loop forever on the following:
 *     Wait until we have work.
 *     Get global values on what work needs to be done.
 *     Call user specified function with argument.
 *     Call barrier code to sync up all threads.
 */static void
idle_threads(thread_p thr)
{
/*
 * initialized the thread thread-private memory pointer.
 */
my_thread_ptr = thr;

    for(;;) {
        /*
         * threads spin here waiting for work to be assign
         * to them.
         */
        while( thr->release_flag == idle_release_flag )
            /* spin until idle_release_flag changes */;

        thr->release_flag = idle_release_flag;
        thr->nsthreads = my_call_thread_max;

        /*
         * call user function with their specified argument.
         */
        if ( thr->stid < my_call_thread_max )
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
        (*my_func)(my_arg);
    /*
     * make all threads join before they were to the idle
     *
     * loop.
    */
my_join_barrier(my_barrier, thr->stid);
}
}
/** create_threads
 * This routine creates all of the MY THREADS package data
 * structures and child threads.
 *
 * arguments:
 *     none
 *
 * return:
 *     none
 *
 * algorithm:
 *     Allocate data structures for a thread
 *     Create the thread via the pthread_create call.
 *     If the create call is successful, repeat until the
 *     number of threads equal the number of processors.
 *
 */

static void
create_threads()
{
    pthread_attr_t attr;
    char *env_val;
    int i, rv, cpus, processors;
    thread_p thr;

    /*
     * allocate and initialize the thread structure for the
     * parent thread.
    */
    if ( (thr = (thread_p) malloc(sizeof(thread_t))) == NULL ) {
        fprintf(stderr, "my_threads: Fatal error: can not
        allocate memory for main thread\n");
        abort();
    }
    my_thread_ptr = thr;

    thr->stid = 0;
    thr->release_flag = 0;

    /*
     * initialize attribute structure
    */
    (void) pthread_attr_init(&attr);

    /*
     * Check to see if the CPS_STACK_SIZE env variable is defined.
    */
}
```



```
    * If it is, then use that as the stacksize.
    */
    if ( (env_val = getenv("CPS_STACK_SIZE")) != NULL ) {
        int val;
        val = atoi(env_val);
        if ( val > 128 )
            my_stacksize = val * K;
    }

(void) pthread_attr_setstacksize(&attr, my_stacksize);

/*
 * determine how many threads we will create.
 */
processors = cpus = pthread_num_processors_np();
if ( (env_val = getenv("MP_NUMBER_OF_THREADS")) != NULL ) {
    int val;

    val = atoi(env_val);
    if ( val >= 1 )
        cpus = val;
}

for(i = 1; i < cpus && i < MAX_THREADS; i++ ) {
    /*
     * allocate and initialize thread data structure.
     */
    if ( (thr = (thread_p) malloc(sizeof(thread_t))) == NULL )
        break;

    thr->stid = i;
    thr->release_flag = 0;

    rv = pthread_create(&my_thread_ids[i-1], &attr,
        (void (*)(void *))idle_threads, (void *) thr);
    if ( rv != 0 ) {
        free(thr);
        break;
    }
    thread_count++;
}

my_threads_are_init = 1;

my_barrier_alloc(&my_barrier, thread_count);

/*
 * since we are done with this attribute, get rid of it.
 */
(void) pthread_attr_destroy(&attr);
}

/*
 * my_ppcall
 * Call user specified routine in parallel.
 */
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
* arguments:
* max- maximum number of threads that are needed.
* func- user specified function to call
* arg- user specified argument to pass to func
*
* return:
* 0- success
* -1- error
*
* algorithm:
* If we are already parallel, then return with an error
* code. Allocate threads and internal data structures,
* if this is the first call.
* Determine how many threads we need.
* Set global variables.
* Signal the child threads that they have parallel work.
* At this point we signal all of the child threads and
* let them determine if they need to take part in the
* parallel call. Call the user specified function.
* Barrier call will sync up all threads.
*/

int
my_ppcall(int max, void (*func)(void *), void *arg)
{
    thread_p thr;
    int i, suspend;

    /*
     * check for error conditions
     */
    if ( max <= 0 || func == NULL )
        return EINVAL;

    if ( my_threads_are_parallel )
        return EAGAIN;

    (void) pthread_mutex_lock(&my_thread_mutex);
    if ( my_threads_are_parallel ) {
        (void) pthread_mutex_unlock(&my_thread_mutex);
        return EAGAIN;
    }

    /*
     * create the child threads, if they are not already created.
     */
    if ( !my_threads_are_init )
        create_threads();

    /*
     * set global variables to communicate to child threads.
     */
    if ( max > thread_count )
        my_call_thread_max = thread_count;
    else
        my_call_thread_max = max;
}
```

```
my_func = func;
my_arg = arg;

my_thread_ptr->nsthreads = my_call_thread_max;

++my_threads_are_parallel;

/*
 * signal all of the child threads to exit the spin loop
 */
++idle_release_flag;

(void) pthread_mutex_unlock(&my_thread_mutex);

/*
 * call user func with user specified argument
 */
(*my_func)(my_arg);

/*
 * call join to make sure all of the threads are done doing
 * there work.
 */
my_join_barrier(my_barrier, my_thread_ptr->stid);

(void) pthread_mutex_lock(&my_thread_mutex);

/*
 * reset the parallel flag
 */
my_threads_are_parallel = 0;

(void) pthread_mutex_unlock(&my_thread_mutex);

return 0;
}

/*
 * my_stid
 * Return thread spawn thread id. This will be in the range
 * of 0 to N-1, where N is the number of threads in the
 * process.
 * arguments:
 * none
 *
 * return
 * spawn thread id
 */

int
my_stid(void)
{
return my_thread_ptr->stid;
}
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
/*
 * my_nsthreads
 *   Return the number of threads in the current spawn.
 *
 * arguments:
 *   none
 *
 * return
 *   number of threads in the current spawn
 */

int
my_nsthreads(void)
{
    return my_thread_ptr->nsthreads;
}

/*
 * my_is_parallel
 *   Return the is parallel flag
 *
 * arguments:
 *   none
 *
 * return
 *   1- if we are parallel
 *   0- otherwise
 */

int
my_is_parallel(void)
{
    int rv;

    /*
     * if my_threads_are_init is set, then we are parallel,
     * otherwise we not.
     */
    (void) pthread_mutex_lock(&my_thread_mutex);
    rv = my_threads_are_init;
    (void) pthread_mutex_unlock(&my_thread_mutex);

    return rv;
}

/*
 * my_complex_cpus
 *   Return the number of threads in the current process.
 *
 * arguments:
 *   none
 *
 * return
 *   number of threads created by this process
 */
```

```
int
my_complex_cpus(void)
{
    int rv;

    /*
     * Return the number of threads that we current have.
     */
    (void) pthread_mutex_lock(&my_thread_mutex);
    rv = thread_count;
    (void) pthread_mutex_unlock(&my_thread_mutex);

    return rv;
}
```

## 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 in that the threads are disjointed.

Some of the asymmetric CPSlib functions map to pthread functions, while others are no longer used, as noted below:

- `cps_thread_create()` spawned asymmetric threads and now maps to the pthread function `pthread_create()`.
- `cps_thread_createn()`, which spawned asymmetric threads with multiple arguments, also maps to `pthread_create()`. However, `pthread_create()` only supports the passing of one argument.
- CPSlib terminated asymmetric threads using `cps_thread_exit()`, which now maps to the pthread function `pthread_exit()`.
- `cps_thread_register_lock` has no corresponding pthread function. It was formerly used in conjunction with `m_lock`, both of which have been replaced with one call to `pthread_join`.
- `cps_plevel()`, the CPSlib function which determined the current level of parallelism, does not have a corresponding pthread function, because levels do not mean anything to pthreads.

The first example in this section `cps_create.c`, provides an example of the above CPSlib functions being used to create asymmetric parallelism.

## Porting CPSlib functions to pthreads

### Using pthreads

#### create.c

```
/*
 * create.c
 *      Show how to use all of the cps asymmetric functions.
 */

#include <cps.h>

mem_sema_t wait_lock;

void
tfunc(void *arg)
{
    int i;

    /*
     * Register the wait_lock, so that the parent thread
     * can wait on us to exit.
     */
    (void) cps_thread_register_lock(&wait_lock);

    for( i = 0; i < 100000; i++ )
        /* spin for a spell */;

    printf("tfunc: ktid = %d\n", cps_ktid());
    cps_thread_exit();
}

main()
{
    int node = 0;
    ktid_t ktid;

    /*
     * Initialize and lock the wait_lock.
     */
    m_init32(&wait_lock, &node);
    m_cond_lock(&wait_lock);

    ktid = cps_thread_create(&node, tfunc, NULL);

    /*
     * We wait for the wait_lock to be release. That is
     * how we know that the child thread
     * has terminated.
     */
    m_lock(&wait_lock);

    exit(0);
}
```

### pth\_create.c

The example below shows how to use the `pth_create.c` function to map to asymmetric functions provided by the CPSlib example.

```
/*
 * pth_create.c
 *      Show how to use all of the pthread functions that
 *      map to cps asymmetric functions.
 *
 */
#include <pthread.h>

void
tfunc(void *arg)
{
    int i;

    for( i = 0; i < 100000; i++ )
        /* spin for a spell */;

    printf("tfunc: ktid = %d\n", pthread_self());
    pthread_exit(0);
}

main()
{
    pthread_t ktid;
    int status;

    (void) pthread_create(&ktid, NULL, (void (*)(void *))
        tfunc, NULL);

    /*
     * Wait for the child to terminate.
     */
    (void) pthread_join(ktid, NULL);

    exit(0);
}
```

## Synchronization using high-level functions

This section demonstrates how to use barriers and mutexes to synchronize symmetrically parallel code.

### Barriers

Implicit barriers are operations in a program where threads are restricted from completion based upon the status of the other threads. For example, in the `ppcall.c` example (on page 311), a join operation occurs after all spawned threads terminate and before the function returns. This type of implicit barrier is often the only type of barrier required.

The `my_barrier.c` example shown below provides a pthreads implementation of CPSlib barrier routines. This includes the following example functions:

- `my_init_barrier` is similar to the `cps_barrier_alloc` function in that it allocates the barrier (`br`) and sets its associated memory counter to zero.
- `my_barrier`, like the CPSlib function `cps_barrier`, operates as barrier wait routine. When the value of the shared counter is equal to the argument  $n$  (number of threads), the counter is set to zero.
- `my_barrier-destroy`, like `cps_barrier_free`, releases the barrier.

```
my_barrier.c
/*
 * my_barrier.c
 *Code to support a fetch and increment type barrier.
 */

#ifdef _HPUX_SOURCE
#define _HPUX_SOURCE
#endif

#include <pthread.h>
#include <errno.h>

/*
 * barrier
 *   magic           barrier valid flag
 *   counter        shared counter between threads
 *   release        shared release flag, used to signal waiting
 *                 threads to stop waiting.
 *   lock           binary semaphore use to control read/write
```



```
*          access to counter and write access to
*          release.
*/

struct barrier {
    int          magic;
    int volatile counter;
    int volatile release;
    pthread_mutex_t lock;
};

#define VALID_BARRIER      0x4242beef
#define INVALID_BARRIER  0xdeadbeef

typedef struct barrier barrier_t;
typedef struct barrier *barrier_p;

/*
 * my_barrier_init
 *   Initialized a barrier for use.
 *
 * arguments
 *   br- pointer to the barrier to be initialize.
 *
 * return
 *   0- success
 *   >0- error code of failure.
 */

int
my_barrier_init(barrier_p *br)
{
    barrier_p b, n;
    int rv;

    b = (barrier_p) *br;

    if ( b != NULL )
        return EINVAL;

    if ( (n = (barrier_p) malloc(sizeof(*n))) == NULL )
        return ENOMEM;

    if ( (rv = pthread_mutex_init(&n->lock, NULL)) != 0 )
        return rv;

    n->magic = VALID_BARRIER;
    n->counter = 0;
    n->release = 0;

    *br = n;

    return 0;
}

/*
```

## Porting CPSlib functions to pthreads

### Using pthreads

```
* my_barrier
*   barrier wait routine.
*
* arguments
*   br           - barrier to wait on
*   n           - number of threads to wait on
*
* return
*   0           - success
*   EINVAL      - invalid arguments
*/
int
my_barrier(barrier_p br, int n)
{
    int rv;
    int key;

    if ( br == NULL || br->magic != VALID_BARRIER )
        return EINVAL;

    pthread_mutex_lock(&br->lock);

    key = br->release; /* get release flag */
    rv = br->counter++; /* fetch and inc shared counter */

    /*
     * See if we are the last thread into the barrier
     */
    if ( rv == n-1 ) {
        /*
         * We are the last thread, so clear the counter
         * and signal the other threads by changing the
         * release flag.
         */
        br->counter = 0;
        ++br->release;
        pthread_mutex_unlock(&br->lock);
    } else {
        pthread_mutex_unlock(&br->lock);

        /*
         * We are not the last thread, so wait
         * until the release flag changes.
         */
        while( key == br->release )
            /* spin */;
    }

    return 0;
}

/*
 * my_barrier_destroy
 * destroy a barrier
 */
```

```
* arguments
*b- barrier to destroy
*
* return
*0- success
*> 0 - error code for why can not destroy barrier
*/

int
my_barrier_destroy(barrier_p *b)
{
    barrier_p br = (barrier_p) *b;
    int rv;

    if ( br == NULL || br->magic != VALID_BARRIER )
        return EINVAL;

    if ( (rv = pthread_mutex_destroy(&br->lock)) != 0 )
        return rv;

    br->magic = INVALID_BARRIER;
    br->counter = 0;
    br->release = 0;

    *b = NULL;

    return 0;
}
```

## Mutexes

Mutexes (binary semaphores) allow threads to control access to shared data and resources. The CPSlib mutex functions map directly to existing pthread mutex functions as shown in Table 61 on page 303. The example below, `pth_mutex.c`, shows a basic pthread mutex program using the `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_trylock`, and `pthread_mutex_unlock`.

There are some differences between the behavior of CPSlib mutex functions and low-level locks (cache semaphores and memory semaphores) and the behavior of pthread mutex functions, as described below:

- CPS cache and memory semaphores do not perform deadlock detection.
- The default pthread mutex does not perform deadlock detection under HP-UX. This may be different from other operating systems. `pthread_mutex_lock` will only detect deadlock if the mutex is of the type `PTHREAD_MUTEX_ERRORCHECK`.

## Porting CPSlib functions to pthreads

### Using pthreads

- All of the CPSlib unlock routines allow other threads to release a lock that they do not own. This is not true with `pthread_mutex_unlock`. If you do this with `pthread_mutex_unlock`, it will result in undesirable behavior.

#### `pth_mutex.c`

```
/*
 * pth_mutex.c
 * Demonstrate pthread mutex calls.
 *
 * Notes when switching from cps mutex, cache semaphore or
 * memory semaphores to pthread mutex:
 *
 *1) Cps cache and memory semaphores did no checking.
 *2) All of the cps semaphore unlock routines allow
 *   other threads to release a lock that they do not
 *   own. This is not the case with
 *   pthread_mutex_unlock. It is either a error or a
 *   undefinedbehavior.
 *3) The default pthread mutex does not do deadlock
 *   detection under HP-UX (this can be different on
 *   other operation systems).
 */

#ifdef _HPUX_SOURCE
#define _HPUX_SOURCE
#endif

#include <pthread.h>
#include <errno.h>

pthread_mutex_t counter_lock;
int volatile counter = 0;

void
tfunc()
{
    (void) pthread_mutex_lock(&counter_lock);
    ++counter;
    (void) pthread_mutex_unlock(&counter_lock);
}

main()
{
    pthread_t tid;

    if ( (errno = pthread_mutex_init(&counter_lock, NULL)) != 0 )
    {
        perror("pth_mutex: pthread_mutex_init failed");
        abort();
    }

    if ( (errno = pthread_create(&tid, NULL, (void (*)(void *))
        tfunc, NULL)) != 0 ) {
```

```
        perror("pth_mutex: pthread_create failed");
        abort();
    }

tfunc();

(void) pthread_join(tid, NULL);

    if ( (errno = pthread_mutex_destroy(&counter_lock)) != 0 ) {
        perror("pth_mutex: pthread_mutex_destroy failed");
        abort();
    }

    if ( counter != 2 ) {
        errno = EINVAL;
        perror("pth_mutex: counter value is wrong");
        abort();
    }

    printf("PASSED\n");
    exit(0);
}
```

## Synchronization using low-level functions

This section demonstrates how to use semaphores to synchronize symmetrically parallel code. This includes functions, such as low-level locks, for which there are pthread mappings, and low-level counter semaphores for which there are no pthread mappings. In this instance, an example is provided so that you can create a program to emulate CPSlib functions, using pthreads.

### Low-level locks

The disposition of CPSlib's low-level locking functions is handled by the pthread mutex functions (as described in Table 61 on page 303). See "Mutexes" on page 327 for an example of how to use pthread mutexes.

### Low-level counter semaphores

The CPSlib [mc]\_init32 routines allocate and set the low-level CPSlib semaphores to be used as counters. There are no pthread mappings for these functions. However, a pthread example is provided below.

This example, `fetch_and_inc.c`, documents the following tasks:

- `my_init` allocates a counter semaphore and initializes the counter associated with it (`p`) to a value.

## Porting CPSlib functions to pthreads

### Using pthreads

- `my_fetch_and_clear` returns the current value of the counter associated with the semaphore and clears the counter.
- `my_fetch_and_inc` increments the value of the counter associated with the semaphore and returns the old value.
- `my_fetch_and_dec` decrements the value of the counter associated with the semaphore and returns the old value.
- `my_fetch_and_add` adds a value (`int val`) to the counter associated with the semaphore and returns the old value of the integer.
- `my_fetch_and_set` returns the current value of the counter associated with the semaphore, and sets the semaphore to the new value contained in `int val`.

The `[mc]_init32` routines allocate and set the low-level cps semaphores to be used as either counters or locks. An example for counters provides pthread implementation in the place of the following CPSlib functions:

- `[mc]fetch32`
- `[mc]_fetch_and_clear32`
- `[mc]_fetch_and_inc32`
- `[mc]_fetch_and_dec32`
- `[mc]_fetch_and_add32`
- `[mc]_fetch_and_set32`

#### **fetch\_and\_inc.c**

```
/*
 * fetch_and_inc
 *   How to support fetch_and_inc type semaphores using pthreads
 */

#ifdef _HPUX_SOURCE
#define _HPUX_SOURCE
#endif

#include <pthread.h>
#include <errno.h>

struct fetch_and_inc {
    int volatilevalue;
    pthread_mutex_tlock;
```

```
};

typedef struct fetch_and_inc fetch_and_inc_t;
typedef struct fetch_and_inc *fetch_and_inc_p;

int
my_init(fetch_and_inc_p *counter, int val)
{
    fetch_and_inc_p p;
    int rv;

    if ( (p = (fetch_and_inc_p) malloc(sizeof(*p))) == NULL )
        return ENOMEM;

    if ( (rv = pthread_mutex_init(&p->lock, NULL)) != 0 )
        return rv;

    p->value = val;

    *counter = p;

    return 0;
}

int
my_fetch(fetch_and_inc_p counter)
{
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}

int
my_fetch_and_clear(fetch_and_inc_p counter)
{
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value;
    counter->value = 0;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}

int
my_fetch_and_inc(fetch_and_inc_p counter)
{

```

## Porting CPSlib functions to pthreads

### Using pthreads

```
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value++;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}

int
my_fetch_and_dec(fetch_and_inc_p counter)
{
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value--;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}

int
my_fetch_and_add(fetch_and_inc_p counter, int val)
{
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value;
    counter->value += val;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}

int
my_fetch_and_set(fetch_and_inc_p counter, int val)
{
    int rv;

    pthread_mutex_lock(&counter->lock);

    rv = counter->value;
    counter->value = val;

    pthread_mutex_unlock(&counter->lock);

    return rv;
}
```



---

## **B**

# **OpenMP Parallel Programming Model**

This appendix discusses HP's native subset implementation of the OpenMP parallel programming model, including OpenMP directives and command line options in the f90 front end and bridge. Topics covered include:

- What is OpenMP?
- HP's implementation of OpenMP
- From HP Programming Model (HPPM) to OpenMP

## What is OpenMP?

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from the desktop to the supercomputer. The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in Fortran on all architectures, including UNIX and Windows NT.

---

## HP's Implementation of OpenMP

HP's native subset implementation of OpenMP includes nine supported directives and four supported data scope clauses, as well as an additional supported clause. This implementation is discussed below.

### OpenMP Command-line Options

The OpenMP directives implemented by HP (and discussed later in this appendix) are only accepted if the new command-line option—`+Oopenmp`—is given. `+Oopenmp` is accepted at all opt levels.

#### Default

The default command line option is `+Onoopenmp`. If `+Oopenmp` is not given, all OpenMP directives (`c$omp`) are ignored.

### OpenMP Directives

This section discusses the implementation of each of the OpenMP directives. In general, work-sharing directives are only accepted at opt level `+03` and above; synchronization directives are accepted at all opt levels. Following is each OpenMP directive and its required opt level:

**Table 63**

**OpenMP Directives and Required Opt Levels**

Directive	Opt Level
PARALLEL	+03
PARALLEL DO	+03
PARALLEL SECTIONS	+03
DO	+03
SECTIONS	+03
SECTION	+03

<b>Directive</b>	<b>Opt Level</b>
CRITICAL	+00
ORDERED	+00
BARRIER	+00

## **OpenMP Data Scope Clauses**

Following are HP's OpenMP supported data scope clauses:

- PRIVATE
- SHARED
- DEFAULT
- LASTPRIVATE

## **Other Supported OpenMP Clauses**

ORDERED

## From HP Programming Model to OpenMP

This section discusses migration from the HP Programming Model (HPPM) to the OpenMP parallel programming model.

### Syntax

The OpenMP parallel programming model is very similar to the current HP Programming Model (HPPM). The general thread model is the same, the spawn (fork) mechanisms behave in a similar fashion, etc. However, the specific syntax to specify the underlying semantics has been changed significantly.

The following table shows the OpenMP directive or clause (relative to the directive) and the equivalent HPPM directive or clause that implements the same functionality. Certain clauses are valid on multiple directives, but are typically listed only once unless there is a distinction warranting further explanation.

Exceptions are defined immediately following the table.

**Table 64**

**OpenMP and HPPM Directives/Clauses**

OpenMP	HPPM
<pre>!\$OMP parallel   private (list)   shared (list)   default (private shared none) </pre>	<pre>!\$dir parallel   task_private(list)   &lt;'shared' is default&gt;   &lt;None, see below&gt; </pre>
<pre>!\$OMP do   schedule(static[,chunkconstant])   ordered </pre>	<pre>!\$dir loop_parallel(dist)   blocked(chunkconstant)   ordered </pre>
<pre>!\$OMP sections </pre>	<pre>!\$dir begin_tasks(dist) </pre>
<pre>!\$OMP section </pre>	<pre>!\$dir next_task </pre>

OpenMP Parallel Programming Model  
 From HP Programming Model to OpenMP

OpenMP	HPPM
!\$OMP parallel do <see parallel and do clauses>	!\$dir loop_parallel <see parallel and loop_parallel(dist) clauses>
!\$OMP parallel sections <see parallel and sections clauses>	!\$dir begin_tasks <see parallel and begin_tasks(dist) clauses>
!\$OMP critical[(name)]	!\$dir critical_section[(name)]
!\$OMP barrier	!\$dir wait_barrier
!\$OMP ordered	!\$dir ordered_section
!\$OMP end_parallel	<none>
!\$OMP end_sections	!\$dir end_tasks
!\$OMP end_parallel_sections	!\$dir end_tasks
!\$OMP end_parallel_do	<none>
!\$OMP end_critical	!\$dir end_critical_section
!\$OMP end_ordered	!\$dir end_ordered_section
!\$OMP end_do	<none>

### Exceptions

- private(list) / loop\_private(list)  
 OpenMP allows the induction variable to be a member of the variable list. HPPM does not.
- default(private|shared|none)

The HPPM defaults to “shared” and allows the user to specify which variables should be private. The HP model does not provide “none”; therefore, undeclared variables will be treated as shared.

- `schedule(static[,chunkconstant]) / blocked([constant])`

Only manifest constants are supported today.

## HP Programming Model Directives

This section describes how the HP Programming Model (HPPM) directives are affected by the implementation of OpenMP.

### Not Accepted with +Oopenmp

These HPPM directives will not be accepted when +Oopenmp is given.

- `parallel`
- `end_parallel`
- `loop_parallel`
- `prefer_parallel`
- `begin_tasks`
- `next_task`
- `end_tasks`
- `critical_section`
- `end_critical_section`
- `ordered_section`
- `end_ordered_section`
- `loop_private`
- `parallel_private`
- `task_private`
- `save_last`
- `reduction`
- `dynsel`

OpenMP Parallel Programming Model  
From HP Programming Model to OpenMP

- barrier
- gate
- sync\_routine
- thread\_private
- node\_private
- thread\_private\_pointer
- node\_private\_pointer
- near\_shared
- far\_shared
- block\_shared
- near\_shared\_pointer
- far\_shared\_pointer

---

**NOTE**

If +Openmp is given, the directives above are ignored.

**Accepted with +Openmp**

These HPPM directives will continue to be accepted when +Openmp is given.

- options
- no\_dynsel
- no\_unroll\_and\_jam
- no\_parallel
- no\_block\_loop
- no\_loop\_transform
- no\_distribute
- no\_loop\_dependence
- scalar
- unroll\_and\_jam
- block\_loop



---

## **More Information on OpenMP**

For more information on OpenMP, see *[www.openmp.org](http://www.openmp.org)*.

OpenMP Parallel Programming Model  
**More Information on OpenMP**

---

# Glossary

**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, which can create dependences 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 and C++ conventions refer to actual arguments as *actual parameters*.

**actual parameter** In C and 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.

**alias** An alternative name for some object, especially an alternative variable name that refers to a memory location.

Aliases can cause data dependences, 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.

**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 performed 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`. In C++, it is done through the dynamic creation of memory blocks using `malloc` or `new`.

---

**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 HP servers 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*.

**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*.

**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 and 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.

**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, or when the bank is not yet finished recycling (i.e., refreshing).

**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 execution past the barrier in sequential mode. On HP servers, 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.

**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. A byte is 8 bits in length.

**cache** A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the 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. HP servers employ separate data and instruction cache memories.

**cache, direct mapped** A form of cache memory that addresses encached data by a function of the data's virtual address. On V2250 servers, the processor cache address is identical to the least-significant 21 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 2 Mbyte (21 bits) apart.

**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 V2250 servers, processor cache lines are 32 bytes

**cache memory** A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the 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. V2250 servers employ separate data and instruction caches.

**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. Cache thrashing also occurs when two or more threads are simultaneously writing to the same cache line.

**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.

**clone** A compiler-generated copy of a loop or procedure. When the HP 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 multiprocessor 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 multiprocessor servers 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.

**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 necessarily 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.

**conventional compiler** A compiler that cannot perform interprocedural optimization.

**counter** A variable that is used to count the number of times an operation occurs.

**CPA** CPU Agent. The gate array on V2250 servers that provides a high-speed interface between pairs of PA-RISC processors and the *crossbar*. Also called the *CPU Agent* and the *agent*.

**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 V2250 servers that provides a high-speed interface between pairs of PA-RISC processors and the *crossbar*.

**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. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses.

**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 V2250 server. Because the crossbar is nonblocking, all ports can run at

---

full bandwidth simultaneously, provided there is not contention for a particular port.

**CSR** Control/Status Register. A CSR is a software-addressable hardware register used to hold control information or state.

**data cache (Dcache)** A small cache memory with a fast access time. This cache holds prefetched and current data. On V2250 servers, processors have 2-Mbyte off-chip caches. See also *cache*, *direct mapped*.

**data dependence** 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 memory accesses.

**data type** A property of a data item that 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. A small cache memory with a one clock cycle access time under pipelined conditions. This cache holds

prefetched and current data. On V2250 servers, this cache is 2 Mbytes.

**deadlock** A condition in which a thread waits indefinitely for some condition or action that cannot, or will not, occur.

**direct memory access (DMA)** A method for gaining direct access to memory and achieving data transfers without involving the CPU.

**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, memory is accessible from the single processor that owns it. Sharing of data requires explicit message passing.

**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 memory and achieving data transfers without involving the CPU.

**double** A double-precision floating-point number that is stored in 64 bits in C and 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 and 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*.

**encache** To copy data or instructions 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.

**fault** A type of *interruption* caused by an instruction requesting a legitimate action that cannot be carried out immediately due to a system problem.

**floating-point** A numerical representation of a real number. On V2250 servers, 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 and 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 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*.

**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.

**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 HP compilers at optimization level +02 and above.

**global register allocation (GRA)** A method by which the compiler attempts to store commonly-referenced scalar variables in registers throughout the code in which they are most frequently accessed.

**global variable** A variable whose scope is greater than a single procedure. In C and 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.

**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 can be exhibited by parallel loops, tasks and expressions, Coarse granularity can be exhibited by parallel processes.

**hand-rolled loop** A loop, more common in Fortran than C or C++, that 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 dependences*, 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.

**HP** Hewlett-Packard, the manufacturer of the PA-RISC chips used as processors in V2250 servers.

**HP-UX** Hewlett-Packard's Unix-based operating system for its PA-RISC workstations and servers.

**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** A set of processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system. Nonscalable servers and V2250 servers consist of one hypernode. When discussing multidimensional parallelism or memory classes, hypernodes are generally called nodes.

**Icache** Instruction cache. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On V2250 servers, this cache is 2 Mbytes.

**IEEE** Institute for Electrical and Electronic Engineers. An international professional organization and a member of ANSI and ISO.

**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
```

**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 (Icache)** This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On V2250 servers, this cache is 2 Mbytes.

**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.

**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, `SIN` is a Fortran intrinsic.

**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.

**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.

**latency** The time delay between the issuing of an instruction and the completion of the operation. A common benchmark used for

comparing systems is the latency of coherent memory access instructions. This particular latency measurement is believed to be a good indication of the *scalability* of a system; low latency equates to low system overhead as system size increases.

**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 to the CPU making the reference.

**local optimization** Restructuring of program statements within the scope of a basic block. Local optimization is done by HP compilers at optimization level +01 and above.

**localization** Data localization. Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly 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 may be larger than the available physical

---

memory. The virtual memory of a V2250 server can be up to 16 Tbytes. HP-UX can map this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called *virtual memory*.

**longword (l)** Doubleword. A primitive data operand which is 8 bytes (64 bits) in length. See also *word*.

**loop blocking** A loop transformation that strip mines 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.

**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** See *induction variable*.

**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, and loop blocking.

**machine exception** A fatal error in the system that cannot be handled by the operating system. See also *exception*.

**main memory** Physical memory other than what the processor caches.

---

**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 and 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.

**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, or when the bank is not yet finished recycling (i.e., refreshing).

**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 Interface (MPI)** A message-passing and process control library. For information on the Hewlett-

Packard implementation of MPI, refer to the *HP MPI User's Guide* (B6011-90001).

**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. MIMD also describes when processes are performing different operations on different data. Compare with SIMD.

**multiprocessing** The creation and scheduling of processes on any subset of CPUs in a system configuration.

**mutex** A variable used to construct an area (region of code) 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.

**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.

**node** On HP scalable and nonscalable servers, a node is equivalent to a *hypernode*. The term “node” is generally used in place of hypernode.

**non-uniform memory access (NUMA)** This term describes memory access times in systems in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results 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. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously provided there is not contention for a particular port.

**NUMA** Non-uniform memory access. This term describes memory access times in systems in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results 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 V2250 servers are 64-bit

values, and must keep address values within a single 16-Tbyte 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. V2250 servers use the 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 idle processor time.

**optimization level** The degree to which source code is optimized by the compiler. The HP compilers offer five levels of optimization: level +00, +01, +02, +03, and +04. The +04 option is not available in Fortran 90.

**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.

**PA-RISC** The Hewlett-Packard Precision Architecture reduced instruction set.

**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 HP-UX servers, the default page size is 4 K (4,096) contiguous bytes. Valid page sizes are: 4 K, 16 K, 64 K, 256 K, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, and 256 Mbytes. See also *virtual memory*.

**page fault** A page fault occurs when a process requests data that is not currently in 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 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.

**parallelization** The process of transforming serial code to a form of code that can run simultaneously on multiple CPUs while preserving semantics. When `+O3 +Oparallel` is specified, the HP compilers automatically parallelize loops in your program and recognize compiler directives and pragmas with which you can manually specify parallelization of loops, tasks, and regions.

**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 dependences 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.

**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. HP programming languages provide compiler directives and pragmas that allow you to identify parallel tasks in source code.

**parameter** In C and 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 within your shell 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 HP-UX supports virtual memory, programs address data by its virtual address; HP-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. V2250 servers can contain up to 16 Gbytes of physical memory on a 16-processor hypernode.

**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," that is, when all stages are operating on separate instructions.

**pipelining** Issuing instructions in an order that best uses the pipeline.

**procedure** A unit of program code. In Fortran, a function, subroutine, or main program; in C and 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.

**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).

**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 dependences 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 V2250 servers employ a RISC architecture.

**reduction** An arithmetic operation that performs a transformation on an array to produce a scalar result.

**reentrancy** The ability of a program unit to be executed by multiple threads at the same time. Each invocation maintains a private copy of its local data and a 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. HP compilers compile for reentrancy by default.

**reference** Any operation that requires a cache line to be encached; this includes load as well as store operations, because 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; because 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 V2250 servers 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.

**scope** The domain in which a variable is visible in source code. The rules that determine scope are different for Fortran and C/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. Pthreads provides a set of manipulation functions to facilitate this.

**shape** The number of elements in each dimension of an array.

**shared virtual memory** A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory.

**shell** An interactive command interpreter that is the interface between the user and the Unix operating system.

**SIMD (single instruction stream multiple data stream)** A computer architecture that performs one operation on multiple sets of data. A processor (separate from the SMP array) is used for the control logic, and the processors in the SMP array perform the instruction on the data. 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 manager or worker 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 Fortran or C.

**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 16 Tbytes in the V2250 servers.

**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, task list, or region 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.

**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). HP-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.

**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 in 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 manager or worker 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 person responsible for managing the administration of a system.

**system manager** The person responsible for the management and operation of a computer system. Also called the system administrator and the sysadmin.

**Tbyte** See terabyte (Tbyte).

**terabyte (Tbyte)**  
1099511627776 ( $2^{40}$ ) bytes.

**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 HP 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).

**translation lookaside buffer** A hardware entity that contains information necessary to translate a virtual memory reference to the corresponding physical page and to validate memory accesses.

**TLB** See *translation lookaside buffer*.

**trip count** The number of iterations a loop executes.

**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.

**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.

**virtual address** The address by which programs access their data. HP-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 V2250 server can be up to 16 Tbytes. The operating system 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*.

**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 V2250

---

servers 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.

**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

- &operator, 31
- +DA, 142
- +DAarchitecture, 141
- +DS, 142
- +DSmodel, 141
- +O[no]aggressive, 114
- +O[no]all, 114, 118
- +O[no]autopar, 114, 118
- +O[no]conservative, 114, 119
- +O[no]dataprefetch, 114, 119
- +O[no]dynsel, 114, 120, 149
- +O[no]entersched, 117, 120
- +O[no]fail\_safe, 114, 121
- +O[no]fastaccess, 114, 121
- +O[no]ftacc, 114, 117, 121
- +O[no]global\_ptrs\_unique, 114, 122, 143
- +O[no]info, 114, 123, 151
- +O[no]initcheck, 115, 117, 123
- +O[no]inline, 55, 57, 91, 92, 112, 115, 124
- +O[no]libcalls, 115, 117, 125
- +O[no]limit, 59, 115, 118, 126
- +O[no]loop\_block, 58, 70, 115, 127, 148
- +O[no]loop\_transform, 58, 70, 79, 82, 84, 89, 115, 127, 148
- +O[no]loop\_unroll, 58, 127
- +O[no]loop\_unroll\_jam, 84, 115, 128, 150
- +O[no]moveflops, 115, 128
- +O[no]multiprocessor, 115, 129
- +O[no]parallel, 94, 115, 149, 160
- +O[no]parmsoverlap, 115, 130
- +O[no]pipeline, 49, 115, 130
- +O[no]procelim, 115, 131
- +O[no]ptrs\_ansi, 115, 131, 143, 267
- +O[no]ptrs\_strongly\_typed, 115, 132, 267
- +O[no]ptrs\_to\_globals, 115, 135, 143
- +O[no]regreassoc, 115, 136
- +O[no]report, 115, 137, 152, 160
- +O[no]sharedgra, 115, 138
- +O[no]signedpointers, 116, 117, 138
- +O[no]size, 59, 116, 138
- +O[no]static\_prediction, 116, 139
- +O[no]vectorize, 116, 117, 139
- +O[no]volatile, 116, 140
- +O[no]whole\_program\_mode, 116, 140
- +O0 optimization, 26
- +O1 optimization, 26
- +O2 optimization, 27, 40, 58
- +O3, 111
- +O3 optimization, 27, 55, 57, 58, 70, 77, 79, 82, 84, 89
- +O4, 111
- +O4 optimization, 55, 57
- +Oinline\_budget, 55, 92, 115, 125
- +Onoinitcheck, 30
- +Oparallel, 111
- +pd, 23
- +pi, 23
- +tmtarget, 141
- [mc]\_fetch\_and\_add32(), 330
- [mc]\_fetch\_and\_clear32(), 330
- [mc]\_fetch\_and\_dec32(), 330
- [mc]\_fetch\_and\_inc32(), 330
- [mc]\_fetch\_and\_set3(), 330
- [mc]\_fetch32(), 330
- [mc]\_init32(), 329

## A

- aC++ compiler
  - location of, 25
  - register allocation, 44
- aC++, parallelism in, 111
- accessing pthreads, 301, 302
- accumulator variables, 281
- actual registers, 40
- address space, virtual, 17
- address-exposed array variables, 144
- addressing, 41
- advanced scalar optimizations, 7
- aggressive optimizations, 118
- algorithm, type-safe, 266
- aliases, 12
  - hidden, 268
  - potential, 267
- aliasing, 59, 64, 69, 266

---

- algorithm, 266
- examples, 64, 65
- mode, 267
- rules, 132
- stop variables, 269
- aliasing rules, type-inferred, 143
- alignment
  - data, 26, 37
  - of arrays, 274
  - simple, 26
- alloc\_barrier functions, 237
- alloc\_gate functions, 237
- alloca(), 125
- ALLOCATE statement, 12, 274
- allocating
  - barriers, 235
  - gates, 235
  - shared memory, 138
  - storage, 204
- allocation functions, 237
- alternate name for object, 64
- Analysis Table, 154, 158
- analysis, flow-sensitive, 269
- ANSI C, 266
  - aliasing algorithm, 266
- ANSI standard rules, 265
- architecture
  - SMP, 1, 2
- architecture optimizations, 141
- arguments
  - block\_factor, 71
  - dummy, 236
- arithmetic expressions, 30, 43, 49, 51, 136
- array, 32
  - address computations, 136
  - address-exposed, 144
  - bounds of, 30
  - data, fetch, 71
  - dimensions, 204
  - indexes, 59
  - references, 31
  - subscript, 106
- arrays
  - access order, 82
  - alignment of, 274
  - dummy arguments, 278
  - equivalencing, 12
  - global, 274
  - LOOP\_PRIVATE, 216
  - of type specifier, 227
  - store, 64
  - strips of, 70
  - unaligned, 278
- asin math function, 126
- assertion, linker disables, 141
- asymmetric parallelism, 321
- asynchronous interrupts, 120
- atan math function, 126
- atan2 math function, 126
- attributes
  - LOOP\_PARALLEL, 181
  - PREFER\_PARALLEL, 181
  - volatile, 32
- automatic parallelism, 94
- avoid loop interchange, 63

**B**

- barrier variable declaration, 235
- barriers, 235, 324
  - allocating, 235
  - deallocating, 238
  - equivalencing, 236
  - high-level, 305
  - wait, 241
- basic blocks, 6
- BEGIN\_TASKS directive and pragma, 94, 177, 192
- block factor, 76
- BLOCK\_LOOP directive and pragma, 70, 76, 146, 148
- blocking, loop, 70
- bold monospace, xvii
- brackets, xvii
  - curly, xvii
- branch

---

---

- destination, 67, 68
- dynamic prediction, 139
- optimization, 26
- static prediction, 139
- branches
  - conditional, 39, 139
  - instruction, 41
  - transforming, 39
  - unconditional, 39
- C**
- C aliasing options, 113
- C compiler
  - location of, 25
  - register allocation, 44
- C compiler option, 283
- cache
  - contiguous, 18
  - data, 12
  - line, 12
  - line boundaries, 275
  - line size, 71
  - lines, fetch, 73
  - lines, fixed ownership, 291
  - padding, 15
  - semaphores, 327
  - thrashing, 13, 78, 271, 290
- cache line boundaries
  - force arrays on (C), 274
  - force arrays on (Fortran), 274
- cache-coherency, 12
- cache-line, 18
- calls
  - cloned, 154, 155
  - inlined, 154, 155
- char, 31
- chatr utility, 23
- check subscripts, 283
- child threads, 309
- CHUNK\_SIZE, 275
- class, 227
  - memory, 223, 225, 226, 227, 228
- clauses, other supported
  - OpenMP, 336
- cloned
  - calls, 154, 155
  - procedures, delete, 140
- cloning, 57, 102, 112
  - across files, 57
  - across multiple files, 112
  - at +O4, 91
  - within files, 57
  - within one source file, 57
- Code, 250
- code
  - contiguous, 197
  - dead, 26
  - entry, 40
  - examining, 294
  - exit, 40
  - isolate in loop, 253
  - loop-invariant, 45
  - motion, 136, 242, 266
  - parallelizing outside loop, 192
  - scalar, 197
  - size, 124
  - synchronizing, 250
  - transformation, 33
- coding
  - guidelines, 30, 31
  - standards, 91
- command syntax, xviii
- command-line options, 55, 115
  - +O[no]\_block\_loop, 70
  - +O[no]\_loop\_transform, 89
  - +O[no]aggressive, 114, 117
  - +O[no]all, 114, 118
  - +O[no]autopar, 114, 118
  - +O[no]conservative, 114, 119
  - +O[no]dataprefetch, 114, 119
  - +O[no]dynsel, 114, 120
  - +O[no]entrysched, 114, 120
  - +O[no]fail\_safe, 114, 121
  - +O[no]fastaccess, 114, 121
  - +O[no]ftacc, 114, 121

---

---

- +O[no]global\_ptrs, 143
- +O[no]global\_ptrs\_unique, 114, 122
- +O[no]info, 114, 123
- +O[no]initcheck, 123
- +O[no]inline, 55, 91, 92, 115, 124
- +O[no]libcalls, 115, 125
- +O[no]limit, 45, 115, 126
- +O[no]loop\_block, 115, 127
- +O[no]loop\_transform, 58, 70, 79, 89, 115, 127
- +O[no]loop\_unroll, 58, 127
  - +O[no]loop\_unroll, 115
- +O[no]loop\_unroll\_jam, 58, 115, 128
- +O[no]moveflops, 115, 128
- +O[no]multiprocessor, 115, 129
- +O[no]parallel, 94, 115, 129
- +O[no]parmsoverlap, 115, 130
- +O[no]pipeline, 49, 115, 130
- +O[no]procelim, 115, 131
- +O[no]ptrs\_ansi, 115, 131, 143, 267
- +O[no]ptrs\_strongly\_typed, 115, 132, 267
- +O[no]ptrs\_to\_globals, 115, 135, 143
- +O[no]regreassoc, 115, 136
- +O[no]report, 115, 137
- +O[no]sharedgra, 115, 138
- +O[no]signedpointers, 116, 138
- +O[no]size, 45, 116, 138
- +O[no]static\_prediction, 116, 139
- +O[no]vectorize, 116, 139
- +O[no]volatile, 116, 140
- +O[no]whole\_program\_mode, 116, 140
- +Oinline\_budget, 55, 92, 115, 125
- +tmtarget, 141
- COMMON, 33
  - blocks, 18, 147, 227, 274
  - statement, 236
  - variable, 91, 150
- common subexpression elimination, 42, 43, 135
- compilation, abort, 121
- compile
  - reentrant, 201
  - time, 44, 126
- compile time, increase, 49
- compiler assumptions, 296
- compiler options
  - C, 283
  - W, 282
- Compiler Parallel Support Library, 301
- compilers
  - location of, 25
  - location of aC++, 25
  - location of C, 25
  - location of Fortran 90, 25
- cond\_lock\_gate functions, 239
- conditional
  - blocks, 197
  - branches, 139
- constant
  - folding, 26
  - induction, 27
- contiguous
  - cache lines, 18
  - code, 197
- control variable, 30
- copy propagation, 135
- core dump, 283
- CPS cache, 327
- cps\_barrier\_free(), 324
- cps\_nsthreads(), 303
- cps\_nthreads(), 310
- cps\_plevel(), 321
- cps\_ppcall(), 310
- cps\_ppcalln(), 310
- cps\_ppcallv(), 303
- CPS\_STACK\_SIZE, 202, 309
- cps\_std(), 303, 310
- cps\_thread\_create(), 321
- cps\_thread\_createn(), 321
- cps\_thread\_exit(), 321
- cps\_thread\_register\_lock(), 321
- CPSlib, 301
  - low-level counter semaphores, 329
  - low-level locking functions, 329
  - unlock routines, 328
  - unmappable functions in pthreads, 310
- CPSlib asymmetric functions, 304
  - cps\_thread\_create(), 304

---

---

cps\_thread\_createn(), 304  
 cps\_thread\_exit(), 304  
 cps\_thread\_register\_lock(), 304  
 cps\_thread\_wait(), 304  
 CPSlib informational functions, 304  
   cps\_complex\_cpus(), 304  
   cps\_complex\_nodes(), 305  
   cps\_complex\_nthreads(), 305  
   cps\_is\_parallel(), 305  
   cps\_plevel(), 305  
   cps\_set\_threads(), 305  
   cps\_topology(), 305  
 CPSlib symmetric functions, 303  
   cps\_nthreads(), 303  
   cps\_ppcall(), 303  
   cps\_ppcalln(), 303  
   cps\_ppcallv(), 303  
   cps\_std(), 303  
   cps\_wait\_attr(), 304  
 CPSlib synchronization functions, 306, 307  
   [mc]\_cond\_lock(), 307  
   [mc]\_fetch\_and\_add32(), 307  
   [mc]\_fetch\_and\_clear32(), 307  
   [mc]\_fetch\_and\_dec32(), 308  
   [mc]\_fetch\_and\_inc32(), 308  
   [mc]\_fetch\_and\_set32(), 308  
   [mc]\_fetch32(), 307  
   [mc]\_free32(), 307  
   [mc]\_init32(), 307, 308  
   cps\_barrier(), 305  
   cps\_barrier\_alloc(), 305  
   cps\_barrier\_free(), 306  
   cps\_limited\_spin\_mutex\_alloc(), 306  
   cps\_mutex\_alloc(), 306  
   cps\_mutex\_free(), 306  
   cps\_mutex\_lock(), 306  
   cps\_mutex\_trylock(), 307  
   cps\_mutex\_unlock(), 307  
 CPU agent, 10  
 create  
   temporary variable, 269  
   threads, 309  
 critical sections, 247  
   conditionally lock, 259  
   using, 250  
 CRITICAL\_SECTION directive and pragma, 177,  
   189, 247  
   example, 190, 250, 252  
 cross-module optimization, 53  
 cumulative optimizations, 58  
 cumulative options, 29  
 curly brackets, xvii

**D**

data

- alignment, 12, 26, 37, 71, 91
- cache, 7, 12, 58, 69, 119
- dependences, 179, 185, 192, 279
- encached, 13
- exploit cache, 102
- item, 228, 231
- items, different, 271
- layout, 271
- local to procedure, 229
- localization, 27, 58, 59, 64, 69
- multiple dependences, 233
- object, 210
- objects (C/C++), 227
- prefetch, 119
- private, 225
- privatizing, 208
- reuse, 12, 13, 71
- segment, 23
- shared, 229
- type statements (C/C++), 235
- types, double, 229

data scope clauses

- OpenMP, 336

DATA statement, 225, 274

data-localized loops, 7

dead code elimination, 26, 40

deadlock, detect with pthreads, 327

deallocating

- barriers, 238
- gates, 235, 250

---

---

deallocation functions, 238  
 default stack size, 175, 202  
 delete
 

- cloned procedures, 140
- inlined procedures, 140

 dependences, 220
 

- data, 179, 185, 192, 279
- element-to-element, 62
- ignore, 149
- loop-carried, 279, 284
- multiple, 233
- nonordered, 247
- ordered data, 233
- other loop fusion, 64
- synchronize, 197
- synchronized, 182
- synchronizing, 248

 dereferences of pointers, 143  
 DIMENSION statement, 236  
 Dipasquale, Mark D., 310  
 directives
 

- BEGIN\_TASKS, 94, 177, 192
- BLOCK\_LOOP, 70, 76, 146, 148
- CRITICAL\_SECTION, 177, 189, 247, 250
- DYNSEL, 146, 148
- END\_CRITICAL\_SECTION, 177, 189, 247
- END\_ORDERED\_SECTION, 248
- END\_PARALLEL, 28, 94, 176
- END\_TASKS, 94, 177, 192
- LOOP\_PARALLEL, 28, 94, 118, 176, 179, 181, 185
- LOOP\_PARALLEL(ORDERED), 245
- LOOP\_PRIVATE, 208, 210
- misused, 284
- NEXT\_TASK, 94, 177, 192
- NO\_BLOCK\_LOOP, 70, 146, 148
- NO\_DISTRIBUTE, 77, 146, 148
- NO\_DYNSEL, 146, 149
- NO\_LOOP\_DEPENDENCE, 60, 63, 149
- NO\_LOOP\_TRANSFORM, 89, 146, 149
- NO\_PARALLEL, 110, 146, 149
- NO\_SIDE\_EFFECTS, 146, 150
- NO\_UNROLL\_AND\_JAM, 85, 146

 OpenMP, 335  
 ORDERED\_SECTION, 177, 248  
 PARALLEL, 94, 176
 

- parallel, 28
- PARALLEL\_PRIVATE, 208, 220
- PREFER\_PARALLEL, 28, 94, 176, 178, 181, 185
- privatizing, 208
- REDUCTION, 146, 177
- SAVE\_LAST, 208, 216
- SCALAR, 146
- SYNC\_ROUTINE, 146, 177, 242
- TASK\_PRIVATE, 196, 208, 218
- UNROLL\_AND\_JAM, 85, 146, 150

 disable
 

- automatic parallelism, 110
- global register allocation, 138
- LCDs, 60
- loop thread parallelization, 191

 division, 40  
 DO loops, 178, 210  
 DO WHILE loops, 184  
 double, 49
 

- data types, 229
- variable, 130, 282

 dummy
 

- argument, 236
- arguments, 278
- registers, 40

 dynamic selection, 120, 154, 155
 

- workload-based, 102, 149

 DYNSEL directive and pragma, 146, 148

**E**

element-to-element dependences, 62  
 ellipses, vertical, xviii  
 encache memory, 20  
 END\_CRITICAL\_SECTION directive and pragma, 177, 189, 247  
 end\_parallel, 28  
 END\_PARALLEL directive and pragma, 28, 94, 176

---

---

END\_TASKS directive and pragma, 94, 177, 192  
 enhance performance, 12  
 entry code, 40  
 environment variables  
   and pthreads, 309  
   CPS\_STACK\_SIZE, 202, 309  
   MP\_IDLE\_THREADS\_WAIT, 100, 309  
   MP\_NUMBER\_OF\_THREADS, 94, 130, 309  
 EQUIVALENCE statement, 64, 266  
 equivalencing  
   barriers, 236  
   gates, 236  
 equivalent groups, constructing, 144  
 ERRNO, 126  
 examining code, 294  
 examples  
   aliasing, 64  
   apparent LCDs, 106  
   avoid loop interchange, 63  
   branches, 40  
   cache padding, 15  
   cache thrashing, 13  
   common subexpression elimination, 43  
   conditionally lock critical sections, 259  
   critical sections and gates, 258  
   CRITICAL\_SECTION, 190, 250  
   data alignment, 37  
   denoting induction variables in parallel loops,  
     213  
   gated critical sections, 252  
   I/O statements, 67  
   inlining with one file, 55  
   inlining within one source file, 55  
   interleaving, 20  
   loop blocking, 76  
   loop distribution, 77  
   loop fusion, 80  
   loop interchange, 82  
   loop peeling, 80  
   loop transformations, 97  
   loop unrolling, 45, 46  
   LOOP\_PARALLEL, 187, 188  
   LOOP\_PARALLEL(ORDERED), 245  
   LOOP\_PRIVATE, 211  
   loop-invariant code motion, 45  
   loop-level parallelism, 94  
   matrix multiply blocking, 74  
   multiple loop entries/exits, 68  
   NO\_PARALLEL, 110  
   node\_private, 231  
   Optimization Report, 160  
   ordered section limitations, 255, 256  
   output LCDs, 106  
   PARALLEL\_PRIVATE, 221  
   parallelizing regions, 199  
   parallelizing tasks, 195, 196  
   PREFER\_PARALLEL, 187, 188  
   reduction, 109  
   SAVE\_LAST, 216  
   secondary induction variables, 214  
   software pipelining, 49  
   strength reduction, 52  
   strip mining, 54  
   SYNC\_ROUTINE, 243, 244  
   TASK\_PRIVATE, 219  
   test promotion, 90  
   thread\_private, 228, 229  
   thread\_private COMMON blocks in parallel  
     subroutines, 229  
   type aliasing, 134  
   unroll and jam, 85  
   unsafe type cast, 133  
   unused definition elimination, 52  
   using LOOP\_PRIVATE w/LOOP\_PARALLEL,  
     211  
 executable files, large, 55, 92  
 execution speed, 130  
 Exemplar Programming Guide for HP-UX  
   systems, 301  
 exit  
   code, 40  
   statement, 68  
 explicit pointer typecast, 144  
 exploit data cache, 102  
 extern variable, 91  
 external, 274

---

---

**F**

- fabs(), 125
- fall-through instruction, 39
- false cache line sharing, 13, 271
- faster register allocation, 40
- file
  - level, 89
  - scope, 31, 227
- file-level optimization, 27
- fixed ownership of cache lines, 291
- float, 49
- float variable, 130
- floating-point
  - calculation, 126
  - expression, 281
  - imprecision, 281
  - instructions, 128
  - traps, 128
- floating-point instructions, 41
- flow-sensitive analysis, 269
- flush to zero, 282
- FMA, 121
- folding, 43, 136
- for loop, 178, 210
- force
  - arrays to start on cache line boundaries (C), 274
  - arrays to start on cache line boundaries (Fortran), 274
  - parallelization, 176, 179
  - reduction, 177
- form of
  - alloc\_barrier, 237
  - alloc\_gate, 237
  - barrier, 235
  - block\_loop, 70
  - cond\_lock\_gate, 239
  - CRITICAL\_SECTION, 247
  - directive names, 147
  - END\_CRITICAL\_SECTION, 247
  - END\_ORDERED\_SECTION, 248
  - free\_barrier, 238
  - free\_gate, 238
  - gate, 235
  - lock\_gate, 239
  - LOOP\_PRIVATE, 210
  - memory class assignments, 226
  - no\_block\_loop, 70
  - no\_distribute, 77
  - no\_loop\_dependence, 60
  - no\_loop\_transform, 89
  - no\_unroll\_and\_jam, 85
  - ORDERED\_SECTION, 248
  - PARALLEL\_PRIVATE, 220
  - pragma names, 147
  - reduction, 108
  - SAVE\_LAST, 216
  - SYNC\_ROUTINE directive and pragma, 242
  - TASK\_PRIVATE, 218
  - unlock\_gate, 240
  - unroll\_and\_jam, 85
- Fortran 90 compiler
  - guidelines, 33
  - location of, 25
- free\_barrier functions, 238
- free\_gate functions, 238
- functions
  - alloc\_barrier, 237
  - alloc\_gate, 237
  - allocation, 237
  - cond\_lock\_gate, 239
  - deallocation, 238
  - free\_barrier, 238
  - free\_gate, 238
  - lock\_gate, 239
  - locking, 239
  - malloc (C), 13, 274
  - memory\_class\_malloc (C), 13, 274
  - number of processors, 203
  - number of threads, 204
  - stack memory type, 205
  - synchronization, 237
  - thread ID, 205
  - unlock\_gate, 240
  - unlocking, 240
  - wait\_barrier, 241
- functions, CPSlib



---

[mc]\_cond\_lock(), 307  
[mc]\_fetch\_and\_add32(), 307, 330  
[mc]\_fetch\_and\_clear32(), 307, 330  
[mc]\_fetch\_and\_dec32(), 308, 330  
[mc]\_fetch\_and\_inc32(), 308, 330  
[mc]\_fetch\_and\_set30(), 330  
[mc]\_fetch\_and\_set32(), 308  
[mc]\_fetch32(), 307, 330  
[mc]\_free32(), 307  
[mc]\_init32(), 307, 308, 329  
asymmetric, 304  
cps\_barrier(), 305  
cps\_barrier\_alloc(), 305  
cps\_barrier\_free(), 306, 324  
cps\_complex\_cpus(), 304  
cps\_complex\_nodes(), 305  
cps\_complex\_nthreads(), 305  
cps\_is\_parallel(), 305  
cps\_limited\_spin\_mutex\_alloc(), 306  
cps\_mutex\_alloc(), 306  
cps\_mutex\_free(), 306  
cps\_mutex\_lock(), 306  
cps\_mutex\_trylock(), 307  
cps\_mutex\_unlock(), 307  
cps\_nthreads(), 310  
cps\_plevel(), 305, 321  
cps\_ppcall(), 303, 310  
cps\_ppcalln(), 303, 310  
cps\_ppcallv(), 303  
cps\_set\_threads(), 305  
cps\_std(), 303, 310  
cps\_thread\_create(), 304, 321  
cps\_thread\_createn(), 304, 321  
cps\_thread\_exit(), 304, 321  
cps\_thread\_register\_lock(), 304, 321  
cps\_thread\_wait(), 304  
cps\_topology(), 305  
cps\_wait\_attr(), 304  
high-level mutexes, 306  
high-level-barriers, 305  
informational, 304  
low-level counter semaphores, 307  
low-level locks, 307

symmetric, 303  
functions, math  
  acos, 126  
  asin, 126  
  atan, 126  
  atan2, 126  
  cos, 126  
  exp, 126  
  log, 126  
  log10, 126  
  pow, 126  
  sin, 126  
  tan, 126  
functions, pthread  
  [mc]\_unlock(), 307  
  pthread\_create(), 304  
  pthread\_exit(), 304  
  pthread\_join(), 304  
  pthread\_mutex\_destroy(), 306  
  pthread\_mutex\_init(), 306, 307, 327  
  pthread\_mutex\_lock(), 306, 307, 327  
  pthread\_mutex\_trylock(), 307, 327  
  pthread\_mutex\_unlock(), 307, 327, 328  
  pthread\_num\_processors\_np(), 304, 305, 311

## G

gate variable declaration, 235  
gates, 147, 189, 235  
  allocating, 235  
  deallocating, 235, 250  
  equivalencing, 236  
  locking, 235  
  unlocking, 235  
  user-defined, 250  
global  
  arrays, 274  
  optimization, 91  
  pointers, 122  
  register allocation, 37, 42, 43, 138  
  variables, 31, 135, 140, 269  
GOTO statement, 39, 67, 68  
GRA, 37, 42, 43, 138

---

guidelines  
  aC++, 30, 31  
  C, 30, 31  
  coding, 31  
  Fortran 90, 30, 33

**H**  
hardware history mechanism, 139  
header file, 124, 226  
hidden  
  aliases, 268  
  ordered sections, 284  
horizontal ellipses, xviii  
HP MPI, 4  
HP MPI User's Guide, 5, 111  
HP-UX Floating-Point Guide, 126, 139, 282  
hypernode, V2250, 11

**I**  
I/O statement, 67  
idle  
  CPS threads, 309  
  threads, 100  
increase replication limit, 87  
incrementing by zero, 296  
induction  
  constants, 27  
  variables, 27, 196  
induction variables, 51, 212, 213, 268  
  in region privatization, 222  
information, parallel, 203  
inhibit  
  data localization, 59  
  fusion, 79  
  localization, 68, 69  
  loop blocking, 76  
  loop interchange, 60, 179  
  parallelization, 266  
inlined calls, 154, 155  
inlined procedures  
  delete, 140  
inlining, 124  
  across multiple files, 92  
  aggressive, 125  
  at +O3, 92  
  at +O4, 92  
  default level, 125  
  within one source file, 55  
inner-loop memory accesses, 82  
instruction  
  fall-through, 39  
  scheduler, 26, 41  
  scheduling, 39, 120  
integer arithmetic operations, 136  
interchange, loop, 63, 68, 77, 82, 90  
interleaving, 17, 18, 19, 20  
interprocedural optimization, 57  
invalid subscripts, 265, 283  
italic, xvii  
iteration  
  distribution, controlling, 273  
  distribution, default, 275  
  stop values, 267  
iterations, consecutive, 245

**K**  
K-Class servers, 9, 225  
kernel parameter, 202  
kernel parameters, 23

**L**  
large trip counts, 299  
LCDs, 59, 279, 284  
  disable, 60  
  output, 106  
levels  
  block, 26  
  optimization, 299  
library calls  
  alloca(), 125  
  fabs(), 125  
  sqrt(), 125  
  strcpy(), 125  
library routines, 126

---

---

limitations, ordered sections, 255, 256  
 linear  
   functions, 51  
   test replacement, 297  
 lint, 31  
 local variables, 31, 209  
 localization, data, 27, 58  
 location of compilers, 25  
 lock\_gate functions, 239  
 locking  
   functions, 239  
   gates, 235  
 locks, low-level, 307  
 log math function, 126  
 logical expression, 36  
 loop, 216  
   arrays, 69  
   blocked, 70  
   blocking, 27, 54, 58, 70, 76, 79, 82, 85, 89, 127, 154, 155  
   blocking, inhibit, 76  
   branch destination, 67  
   counter, 268  
   customized, 213  
   dependence, 149  
   disjoint, 99  
   distribution, 27, 58, 70, 79, 82, 85, 89, 127, 154, 155  
   distribution, disable, 148  
   entries, extra, 68  
   entries, multiple, 59  
   fused, 157, 162  
   fusion, 27, 58, 70, 79, 80, 82, 89, 127, 155  
   fusion dependences, 59, 64  
   induction, 181  
   induction variable, 196  
   interchange, 27, 58, 67, 68, 69, 70, 76, 77, 79, 82, 85, 89, 90, 154, 155  
   interchange, avoid, 63  
   interchange, inhibit, 60, 179  
   interchanges, 150  
   invocation, 185  
   iterations, 271  
   jamming, 128  
   multiple entries in, 68  
   nest, 45, 76  
   nested, 20, 84, 85  
   nests, 153  
   number of, 104  
   optimization, 53  
   optimize, 149  
   overhead, eliminating, 128  
   parallelizing, 212  
   peeled iteration of, 80  
   peeling, 80, 155  
   preventing, 28  
   promotion, 155  
   reduction, 157  
   relocate, 82  
   removing, 157  
   reordering, 28, 89  
   replication, 45, 58  
   restrict execution, 182  
   serial, 20, 183  
   source line of, 159  
   strip length, 54  
   table, 159  
   thread parallelization, 191  
   transformations, 7, 58, 82, 97  
   unroll, 45, 79, 82, 84, 89, 127  
   unroll and jam, 28, 54, 58, 79, 82, 84, 89, 127, 154, 155  
   unroll factors, 87  
   unroll\_and\_jam, 70  
   unrolling, 42, 45, 46, 58, 128  
 Loop Report, 137, 151, 153, 159  
 loop unrolling example, 45  
 loop, strip, 72  
 LOOP\_PARALLEL, 181  
 loop\_parallel, 28  
 LOOP\_PARALLEL directive and pragma, 28, 94, 118, 129, 176, 179, 185  
   example, 187, 188, 213  
 LOOP\_PARALLEL(ORDERED) directive and pragma, 245, 287  
   example, 245

---

---

LOOP\_PRIVATE directive and pragma, 208, 210  
   arrays, 216  
   example, 211  
 loop-carried dependences, 59, 60, 279, 284  
 loop-invariant, 46  
   code, 42, 45  
   code motion, 136  
 loop-iteration count, 102  
 loops  
   adjacent, 80  
   constructing, 30  
   data-localized, 7  
   DO (Fortran), 178, 210  
   DO WHILE (Fortran), 184  
   exploit parallel code, 246  
   for (C), 178, 210  
   fusible, 79  
   fusing, 150  
   induction variables in parallel, 212  
   multiple entries, 68  
   neighboring, 79  
   number of parallelizable, 79  
   parallelizing, 175  
   parallelizing inner, 290  
   parallelizing outer, 290  
   privatization for, 159  
   privatizing, 207  
   reducing, 79  
   replicated, 90  
   safely parallelizing, 267  
   simple, 102  
   that manipulate variables, 207  
   triangular, 188, 288  
   unparallelizable, 180  
 loop-variant, 46  
 low-level  
   counter semaphores, 307, 329  
 LSIZE, 278

**M**

machine  
   instruction optimization, 26  
   instructions, 84  
   loading, 96  
 MACs, 10  
 malloc, 12, 274  
 man pages, xviii  
 Managing Systems and Workgroups, 202  
 manual  
   parallelization, 179, 208  
   synchronization, 208, 258  
 map-coloring, 44  
 Mark D. Dipasquale, 310  
 math functions, 126  
 matrix multiply blocking, 74  
 memory  
   banks, 10  
   encached, 20  
   hypernode local, 223  
   inner-loop access, 82  
   layout scheme, 32  
   mapping, 33  
   overlap, 130  
   physical, 17  
   references, 140  
   semaphores, 327  
   space, occupying same, 267  
   usage, 126  
   virtual, 18, 46  
 Memory Access Controllers, 10  
 memory class, 208, 228  
   assignments, 226  
   declarations (C/C++), 226  
   declarations (Fortran), 226  
   misused, 265  
   node\_private, 223, 225, 231  
   thread\_private, 223, 225  
 memory\_class\_malloc, 12, 13, 274  
 message-passing, 4  
 minimum page size, 23  
 misused  
   directives and pragmas, 284  
   memory classes, 265  
 monospace, xvii  
 MP\_IDLE\_THREADS\_WAIT, 100, 309

---

---

MP\_NUMBER\_OF\_THREADS, 94, 130, 309  
MPI, 4  
multinode servers, 301  
multiple  
  data dependences, 233  
  entries in loop, 68  
  exits, 69  
multiplication, 40  
mutexes, 324, 327  
  high-level, 306

## N

natural boundaries, 37  
nested  
  loop, 20  
  parallelism, 234  
NEXT\_TASK directive and pragma, 94, 177, 192  
NO\_BLOCK\_LOOP directive and pragma, 70,  
  146, 148  
NO\_DISTRIBUTE directive and pragma, 77, 146,  
  148  
NO\_DYNSEL directive and pragma, 146, 149  
NO\_LOOP\_DEPENDENCE directive and  
  pragma, 60, 63, 149, 286  
  directives  
    NO\_LOOP\_DEPENDENCE, 146  
NO\_LOOP\_TRANSFORM directive and pragma,  
  89, 146, 149  
NO\_PARALLEL directive and pragma, 110, 146,  
  149  
NO\_SIDE\_EFFECTS directive and pragma, 146,  
  150  
NO\_UNROLL\_AND\_JAM directive and pragma,  
  85, 146  
NO\_UNROLL\_JAM directive and pragma, 84  
node\_private, 111  
  example, 231  
  static assignment of, 228, 231  
  virtual memory class, 223, 225  
nondeterminism of parallel execution, 284, 287  
nonordered  
  dependences, 247

  manipulations, 177  
  nonstatic variables, 33, 123  
  Norton, Scott, 310  
  notational conventions, xvii  
  number of  
    processors, 129, 203  
    threads, 204

## O

O, 143  
objects, stack-based, 227  
offset indexes, 278  
OpenMP, 334  
  clauses, other supported, 336  
  Command-line Options, 335  
    default, 335  
  data scope clauses, 336  
  defined, 334  
  directives, 335  
  Directives and Required Opt Levels, 335  
  effect on HPPM directives, 339  
  HP's implementation of, 335  
  More information, 341  
  syntax, 337  
  www.openmp.org, 341  
operands, 36  
optimization, 26  
  +O0, 26  
  +O1, 26  
  +O2, 27, 40, 58  
  +O3, 27, 55, 57, 58, 70, 77, 79, 82, 84, 89  
  +O4, 55, 57  
  aliasing, 64  
  block-level, 26, 39  
  branch, 26, 39  
  cloning within one file, 57  
  command-line options, 26, 93  
  cross-module, 53, 91  
  cumulative, 58  
  data localization, 58, 69  
  dead code, 39  
  directives, 113

---

faster register allocation, 39  
 features, 26, 35, 53  
 file-level, 27  
 FMA, 122  
 global, 91  
 I/O statements, 67  
 inlining across multiple files, 92  
 inlining within one file, 55  
 interprocedural, 57, 112  
 levels, 25, 266, 299  
 loop, 53  
 loop blocking, 70  
 loop distribution, 77  
 loop fusion, 79  
 loop interchange, 82  
 loop unroll and jam, 84  
 multiple loop entries, 68  
 multiple loop exits, 68  
 options, 113  
 peephole, 26, 39, 41  
 pragmas, 113  
 routine-level, 27, 42  
 static variable, 91  
 store/copy, 27  
 strip mining, 54  
 test promotion, 90  
 unit-level, 6  
 using, 30  
 valid options, 114  
 Optimization Report, 85, 90, 151, 158, 183  
   contents, 137  
 Optimization Reports, 267  
 optimizations  
   advanced, 7  
   advanced scalar, 7  
   aggressive, 118  
   architecture-specific, 141  
   floating-point, 121  
   increase code size, 138  
   loop reordering, 89  
   scalar, 6, 7  
   suppress, 138  
   that replicate code, 87  
   optimize  
     instruction scheduling, 120  
     large programs, 139  
     loop, 149  
   ordered  
     data dependences, 233  
     parallelism, 194, 246  
     sections, 248  
   ordered sections  
     hidden, 284  
     limitations of, 255, 256  
     using, 253  
 ORDERED\_SECTION directive and pragma, 177, 248  
 output LCDs, 106  
 overflowing trip counts, 297  
 overlap, memory, 130

**P**

PA-8200, 23  
 page size, minimum, 23  
 parallel  
   assignments, 44  
   command-line options, 93  
   construct, 247  
   executables, 12  
   execution, 287  
   information functions, 175, 203  
   programming, 9  
   programming techniques, 175  
   regions, 176  
   structure, 234  
   synchronization, 233  
   tasks, 177  
   threads, 138  
 PARALLEL directive and pragma, 94, 176  
 PARALLEL\_PRIVATE directive and pragma, 208, 220  
   example, 221  
 parallelism, 29, 110  
   asymmetric, 321  
   automatic, 94

---

---

- in aC++, 111
- inhibit, 28
- levels of, 94
- loop level, 94
- nested, 234
- ordered, 194, 246
- region level, 94
- stride-based, 186
- strip-based, 99, 186
- task level, 94
- thread, 234
- unordered, 193
- parallelization, 28, 54
  - force, 176, 179
  - in aC++, 28
  - increase, 178
  - inhibit, 266
  - manual, 179, 208
  - overhead, 291
  - prevent, 28
  - preventing, 110
- parallelizing
  - code outside a loop, 192
  - consecutive code blocks, 177
  - inner loops, 290
  - loop, 212
  - loops, safely, 267
  - next loops, 178
  - outer loops, 290
  - regions, 197
  - tasks, 192
  - threads, 183, 191
- parameters, kernel, 23
- partial evaluation, 36
- PCI bus controller, 10
- peephole optimization, 26, 41
- performance
  - enhance, 12
  - shared-memory programs, 208
- physical memory, 17
- pipelining, 41
- prerequisites, 49
  - software, 49
- pointers, 31
  - C, 266
  - dereferences, 143
  - strongly-typed, 132
  - type-safe, 132
  - using as loop counter, 268
- poor locality, 139
- porting
  - CPSlib functions to pthreads, 301
  - multinode applications, 225
  - X-Class to K-Class, 224
  - X-Class to V-Class, 224
- POSIX threads, 111, 301
- potential alias, 267
- pow math function, 126
- pragmas
  - begin\_tasks, 94, 177, 192
  - block\_loop, 70, 76, 146, 148
  - critical\_section, 177, 189, 247
  - critical\_section, 250
  - dynsel, 146, 148
  - end\_critical\_section, 177, 189, 247
  - end\_ordered\_section, 248
  - end\_parallel, 28, 94, 176
  - end\_tasks, 94, 177, 192
  - loop\_parallel, 28, 94, 118, 176, 179, 181, 185
  - loop\_parallel(ordered), 245
  - loop\_private, 208, 210
  - misused, 284
  - next\_task, 94, 177, 192
  - no\_block\_loop, 70, 146, 148
  - no\_distribute, 146, 148
  - no\_dynsel, 146, 149
  - no\_loop\_dependence, 60, 146, 149
  - no\_loop\_transform, 89, 146, 149
  - no\_parallel, 110, 146, 149
  - no\_side\_effects, 146, 150
  - no\_unroll\_and\_jam, 85, 146
  - ordered\_section, 177, 248
  - parallel, 28, 94, 176
  - parallel\_private, 208, 220
  - prefer\_parallel, 28, 94, 176, 178, 181, 185
  - privatizing, 208

---

---

- reduction, 146, 177
- save\_last, 208, 216
- scalar, 146
- sync\_routine, 44, 146, 177, 242
- task\_private, 196, 208, 218
- unroll\_and\_jam, 85, 146, 150
- prefer\_parallel, 182
- PREFER\_PARALLEL directive and pragma, 28, 94, 129, 176, 178, 181, 185
  - example, 187, 188
- prevent
  - loop interchange, 67
  - parallel code, 149
  - parallelism, 110
- primary induction variable, 184
- private data, 225
- privatization
  - data, 185
  - variable, 159
- Privatization Table, 137, 152, 159
- privatizing
  - directives, 208
  - loop data, 210
  - loops, 159
  - parallel loops, 208
  - pragmas, 208
  - regions, 208, 220
  - tasks, 208, 218
  - variables, 208
- procedure calls, 59, 266
- procedures, 6
- processors
  - number of, 203
  - specify number of, 129
- program
  - behavior, 120
  - overhead, 248, 249, 291
  - units, 6
- programming models
  - message-passing, 4
  - shared-memory, 3
- programming parallel, 9
- propagation, 43
- prototype definition, 125
- pthread
  - mutex functions, 327
  - mutexes, 329
- pthread asymmetric functions
  - pthread\_create(), 304
  - pthread\_exit(), 304
  - pthread\_join(), 304
- pthread informational functions
  - pthread\_num\_processors\_np(), 304, 305
- pthread synchronization functions
  - [mc]\_unlock(), 307
  - pthread\_mutex\_destroy(), 306
  - pthread\_mutex\_init(), 306, 307
  - pthread\_mutex\_lock(), 306, 307
  - pthread\_mutex\_trylock(), 307
  - pthread\_mutex\_unlock(), 307
- pthread.h, 302
- pthread\_mutex\_init(), 327
- pthread\_mutex\_lock(), 327
- pthread\_mutex\_trylock(), 327
- pthread\_mutex\_unlock(), 327, 328
- pthreads, 111, 301
  - accessing, 301, 302
  - and environment variables, 309

**R**

- REAL variable, 130
- REAL\*8 variable, 130, 282
- reduction
  - examples, 109
  - force, 177
  - form of, 108
  - loop, 157
- REDUCTION directive and pragma, 146, 177
- reductions, 28, 281, 284, 286
- reentrant compilation, 175, 201
- region privatization, induction variables in, 222
- regions
  - parallelizing, 175, 197
  - parallelizing, example, 199
  - privatizing, 207, 220

---



---

register  
   allocation, 44  
   allocation, disable, 138  
   exploitation, 128  
   increase exploitation of, 84  
   reassociation, 46  
   usage, 79  
   use, improved, 128  
 registers, 26, 51  
   global allocation, 37, 42, 43  
   simple alignment, 37  
 reordering, 154  
 replicate code, 87  
 replication limit, increase, 87  
 report\_type, 137, 152  
 report\_type values  
   all, 152  
   loop, 152  
   none, 152  
   private, 152  
 RETURN statement, 59, 68  
 return statement, 59, 68  
 reuse  
   spatial, 71, 74  
   temporal, 71, 74, 84  
 routine-level optimization, 27, 42  
 routines  
   user-defined, 242  
   vector, 139  
 rules  
   ANSI standard, 265  
   scoping, 231

**S**

SAVE variable, 91  
 SAVE\_LAST directive and pragma, 208, 216  
   example, 216  
 scalar  
   code, 197  
   optimizations, 6, 7  
   variables, 43, 277  
 SCALAR directive and pragma, 146  
 scheduler, instruction, 41  
 scope of this manual, xvi  
 scoping rules, 231  
 Scott Norton, 310  
 secondary induction variables, 213  
   example, 214  
 semaphores  
   binary, 327  
   low-level, 307  
   low-level counter, 329  
 serial  
   function, 20  
   loop, 183  
 servers  
   K-Class, 9, 141  
   V2250, 9, 141  
   V-Class, 9, 141  
 shared  
   data, 4  
   variable, 177  
 shared-memory, 3  
 shared-memory programs, optimize, 223  
 short, 31  
 short-circuiting, 36  
 signed/unsigned type distinctions, 144  
 simple loops, 102  
 sin math function, 126  
 single-node servers  
   porting multinode apps to, 225  
 SMP  
   architecture, 1, 2  
   software pipelining, 27, 42, 49, 130, 136  
   space, virtual address, 17  
   spatial reuse, 71, 74  
 spawn  
   parallel processes, 4  
   thread ID, 96  
   threads, 209  
 speed, execution, 130  
 spin  
   suspend, 309  
   wait, 309  
 spp\_prog\_model.h, 203, 226

---

---

sqrt(), 125  
 stack  
   memory type, 205  
   size, default, 202  
 stack-based objects, 227  
 statements  
   ALLOCATE (Fortran), 13, 274  
   COMMON (Fortran), 236  
   DATA (Fortran), 225, 274  
   DIMENSION (Fortran), 236  
   EQUIVALENCE (Fortran), 64, 266  
   exit (C/C++), 68  
   GOTO (Fortran), 67, 68  
   I/O (Fortran), 67  
   return (C/C++), 59, 68  
   RETURN (Fortran), 59, 68  
   stop (C/C++), 59  
   STOP (Fortran), 59, 68  
   throw (C++), 69  
   type, 236  
 static  
   variables, 33, 91  
 static assignments  
   node\_private, 228, 231  
   thread\_private, 228  
 STOP statement, 59, 68  
 stop statement, 59  
 stop variables, 269  
 storage class, 227  
   external, 274  
 storage location  
   of global data, 91  
   of static data, 91  
 strcpy(), 125  
 strength reduction, 27, 51, 136  
 stride-based parallelism, 186  
 strip mining, 54, 97  
   example, 54  
   length, 72  
 strip-based parallelism, 99, 186  
 strip-mining, 7  
 strlen(), 125  
 strongly-typed pointers, 132  
  
 structs, 31, 274  
 structure type, 144  
 subroutine call, 155  
 sudden underflow, enabling, 282  
 sum operations, 109  
 suppress optimizations, 138  
 suspend wait, 309  
 sync\_routine, 44, 242  
 SYNC\_ROUTINE directive and pragma, 146, 177  
   example, 243, 244  
 synchronization  
   functions, 237  
   intrinsic, 245  
   manual, 208, 258  
   parallel, 233  
   using high-level barriers, 305  
   using high-level mutexes, 306  
   using low-level counter semaphores, 307  
 synchronize  
   code, 250  
   dependences, 197  
   symmetrically parallel code, 324  
 syntax  
   OpenMP, 337  
 syntax extensions, 226  
 syntax, command, xviii  
  
**T**  
 tan math function, 126  
 TASK\_PRIVATE directive and pragma, 196, 208,  
   218  
   example, 219  
 tasks  
   parallelizing, 175, 177, 192  
   parallelizing, example, 195, 196  
   privatizing, 207, 218  
 Tbyte, 4  
 temporal reuse, 71, 74, 84  
 terabyte, 4  
 test  
   conditions, 26  
   promotion, 28, 90, 154

---

---

text segment, 23  
THEN clause, 39  
thrashing, cache, 290  
thread, 148  
    affinity, 100  
    ID, 205, 234  
    ID assignments, 234  
    idle, 96  
    noidle, 96  
    spawn ID, 96  
    stack, 205  
    suspended, 100  
    waking a, 100  
thread\_private, 111  
    example, 228, 229  
    static assignment of, 228  
    virtual memory class, 223, 225  
thread\_trip\_count, 104  
thread-parallel construct, 234  
threads, 96  
    child, 309  
    create, 309  
    idle, 100, 309  
    number of, 204  
    parallelizing, 183, 191  
    spawn parallel, 102  
    spawned, 209  
thread-specific array elements, 276  
Threadtime, 310  
threshold iteration counts, 104  
throw statement, 59, 69  
time, 118  
transformations, 39  
    loop, 97  
    reordering, 149  
triangular loops, 188, 288  
trip counts  
    large, 299  
    overflowing, 297  
type  
    aliasing, 134, 136  
    casting, 132  
    names, synonymous, 144

    specifier, 227  
    statements, 236  
    union, 144  
type-checking, 266  
type-incompatible assignments, 145  
type-inferred aliasing rules, 143  
type-safe  
    algorithm, 266  
    pointers, 132

## U

unaligned arrays, 278  
uninitialized variables, 123  
union type, 144  
unlock\_gate function, 240  
unlocking  
    functions, 240  
    gates, 235  
unordered parallelism, 193  
unparallelizable loops, 180  
Unroll and Jam, 156  
unroll and jam, 28  
    automatic, 128  
    directive-specified, 128  
unroll factors, 46, 87  
UNROLL\_AND\_JAM directive and pragma, 85,  
    146, 150  
unrolling, excessive, 87  
unsafe type cast, 133  
unused definition elimination, 52  
using  
    a pointer as a loop counter, 268  
    critical sections, 250  
    hidden aliases as pointers, 268  
    ordered sections, 253

## V

V2250 servers, 9, 71, 141, 223  
    chunk size, 295  
    hypernode overview, 11  
valid page sizes, 23  
variables

---

accumulator, 281  
char, 31  
COMMON (Fortran), 33, 91, 150  
create temporary, 269  
double (C/C++), 130, 282  
extern (C/C++), 91  
float (C/C++), 130  
global, 31, 135, 140, 269  
induction, 27, 45, 213, 222, 268  
iteration, 45  
local, 30, 31, 33, 209  
loop induction, 181  
nonstatic, 33, 123  
primary induction, 184  
privatizing, 159, 185, 208  
REAL (Fortran), 130  
REAL\*8 (Fortran), 130, 282  
register, 31  
SAVE (Fortran), 91  
scalar, 37, 43, 277  
secondary induction, 213  
secondary induction, example, 214  
shared, 177, 225  
shared-memory, 138  
short, 31  
static, 33, 123  
static (C/C++), 91  
stop, 269  
uninitialized, 123  
values of, 36  
V-Class Architecture manual, 9  
V-Class servers, 9, 225  
  hypernode overview, 11  
vector routines, 139, 140  
vertical ellipses, xviii  
virtual  
  address space, 17  
  memory, 18  
  memory address, 46  
volatile attribute, 32  
vps\_ceiling, 23  
vps\_chatr\_ceiling, 23  
vps\_pagesize, 23

## W

-W compiler option, 282  
wait\_barrier functions, 241  
workload-based dynamic selection, 102, 149

## X

X-class, 224