

Exemplar C and Fortran 77 Programmer's Guide

for HP-UX Systems

First Edition



B6057-96002

Exemplar Fortran 77, Exemplar C

Customer Order Number: B6057-90002

December 1997

Notice

© Copyright Hewlett-Packard Company 1997. 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

How to use this guide	ix
Purpose and audience	ix
Notational conventions	ix
Notes	x
Associated documents	xi
Ordering documents	xii
1 Introduction	1
The programming model	1
Standard HP compiler information	2
+O0 (default)	2
+O1	3
+O2, -O	3
+O3	5
+O4	5
+O[no]aggressive	6
+O[no]all	6
+O[no]fail_safe	6
+O[no]info	7
+Oinline_budget= <i>n</i>	7
+O[no]limit	7
+O[no]loop_transform	8
+O[no]loop_unroll[= <i>n</i>]	8
+O[no]size	8
Compiler usage	9
Using the C compiler	9
Using the Fortran 77 compiler	10
Options to get you started	11

2 Exemplar extensions	13
Exemplar compiler options	14
-g	14
-I8	14
+O[no]autopar	15
+O[no]dataprefetch	15
+O[no]dynsel	16
+O[no]exemplar_model	16
+O[no]loop_block	17
+O[no]loop_unroll_jam	17
+O[no]parallel	18
+O[no]report[= <i>report_type</i>]	19
+O[no]sharedgra	19
+pa	20
+pal	20
+tm <i>target</i>	20
Exemplar compiler directives and pragmas	22
align_cti(<i>namelist</i>)	23
barrier(<i>namelist</i>)	23
begin_tasks[(<i>attribute_list</i>)]	24
block_loop[(block_factor= <i>n</i>)]	25
block_shared(<i>allocatable_array_namelist</i>)	25
critical_section[(<i>gate_var</i>)]	25
dynsel[(<i>trip_count=n</i>)]	25
end_critical_section	26
end_ordered_section	26
end_parallel	26
end_tasks	26
far_shared(<i>namelist</i>)	26
far_shared_pointer(<i>namelist</i>)	27
gate(<i>namelist</i>)	27
loop_parallel[(<i>attribute_list</i>)]	27
loop_private(<i>namelist</i>)	29
near_shared(<i>namelist</i>)	29
near_shared_pointer(<i>namelist</i>)	29
next_task	30
no_block_loop	30
no_distribute	30
no_dynsel	30
no_loop_dependence(<i>namelist</i>)	30
no_loop_transform	31
no_parallel	31
no_side_effects(<i>funclist</i>)	31
no_unroll_and_jam	31

node_private(<i>namelist</i>)	31
node_private_pointer(<i>namelist</i>)	32
ordered_section(<i>gate_var</i>)	32
parallel[(<i>attribute_list</i>)]	33
parallel_private(<i>namelist</i>)	33
prefer_parallel[(<i>attribute_list</i>)]	34
reduction(<i>namelist</i>)	35
save_last[(<i>list</i>)]	35
scalar	35
sync_routine(<i>routinelist</i>)	36
task_private(<i>namelist</i>)	36
thread_private(<i>namelist</i>)	36
thread_private_pointer(<i>namelist</i>)	36
unroll_and_jam[(unroll_factor= <i>n</i>)]	37
Exemplar Fortran 77 language extensions	37
INTEGER*8	37
INTEGER*8 constants	37
LOGICAL*8	37
TASK COMMON	38
Exemplar Fortran 77 intrinsics	39
Exemplar Fortran 77 equivalences	40
Predefined symbols	41
3 Developing parallel applications	43
Compiler-generated parallelism	44
Using the +Oparallel compiler option	44
HP MPI	45
Pthreads	46
Accessing Pthreads	46
The Compiler Parallel Support library (CPSlib)	47
Accessing CPSlib	47
4 The Exemplar assembler and linker	49
The assembler	49
Assembler usage	50
The linker	51
Object file formats: SOM and ELF	51
Linking to debug or profile	51
Linker usage	52

5 Debugging and profiling	53
The CXdb debugger	54
Using CXdb	55
The CXperf profiler	56
Using CXperf	57
Appendix A: Environment variables	59
CCOPTS	60
FCOPTS	62
MP_NUMBER_OF_THREADS	64
TMPDIR	64
Index	65

Tables

Table 1	Optimizations performed at +O0	.2
Table 2	Optimizations performed at +O1	.3
Table 3	Optimizations performed at +O2	.4
Table 4	Options to get you started	.11
Table 5	+tm <i>target</i> and +DA/+DS	.21
Table 6	Intrinsic functions	.39

How to use this guide

Purpose and audience

This guide describes how to use the Hewlett-Packard Exemplar C and Fortran 77 compilers. It describes the differences between the Exemplar compilers and the standard Hewlett-Packard (HP) compilers on which they are based. All Exemplar-specific features are described in detail.

The audience for this book is the experienced C or Fortran 77 programmer who has a basic familiarity with HP-UX or Unix and is using the Exemplar C compiler V1.2.3 (or higher) or the Exemplar Fortran 77 compiler V1.2.3 (or higher) to produce applications for HP-UX 11.0 (or higher) running on a K-Class or V-Class server.

Notational conventions

This section discusses notational conventions used in this book.

Bold monospace

In command examples, text shown in **bold monospace** identifies user input that must be typed exactly as shown.

Monospace

In paragraph text, `monospace` identifies command names, system calls, and directive/pragma names.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

{ }

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:

```
command {a | b}
```

[]

In command syntax diagrams and directive/pragma specifications, square brackets indicate optional data.

The following command example indicates that the variable *output_file* is optional:

```
command input_file [output_file]
```

...

In command syntax, horizontal ellipses show repetition of the preceding item(s).

The following command example indicates you can optionally specify more than one *input_file* on the command line:

```
command input_file [input_file ...]
```

Notes

This document presents notes in the following format:

NOTE

A Note highlights supplemental information.

Associated documents

Hewlett-Packard Company provides the following documents to help you use the C and Fortran 77 compilers and associated tools:

- *Programming on HP-UX* (B2355-90652)—This book describes how to develop software on HP-UX using HP compilers, assemblers, linker, libraries, and object files.
- *HP C/HP-UX Reference Manual* (92453-90024)—This manual presents reference information on the C programming language, as implemented by Hewlett-Packard.
- *HP C/HP-UX Programmer's Guide* (92434-90002)—This guide contains detailed discussions of selected C topics.
- *FORTRAN/9000 Programmer's Reference* (B3906-90002)—This book is a Hewlett-Packard Fortran 77 language reference.
- *FORTRAN/9000 Programmer's Guide* (B3906-90001)—This manual is a task reference. It describes features and requirements in terms of the tasks a programmer might perform. These tasks include how to compile, link, run, debug, and optimize programs.
- *CXdb Quick Reference* (B5639-90001)—This book covers the more frequently used features of the CXdb visual debugger.
- *Exemplar Programming Guide for HP-UX Systems* (B6056-90002)—This book describes efficient shared-memory programming techniques for the K-Class and V-Class servers.
- *HP MPI User's Guide* (B6011-90001)—This book discusses message-passing programming using Hewlett-Packard's Message-Passing Interface library.

- *HP Fortran 90 Programmer's Reference* (B5876-90001)—This book is a complete Fortran 90 language reference. It also covers compiler options, compiler directives, and library information.
- *HP Fortran 90 Programmer's Guide* (B6056-90003)—This book 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 from HP Fortran 90.
- *Assembly Language Reference Manual* (92432-90001)—This manual describes the use of the Precision Architecture RISC (PA-RISC) Assembler.
- The following man pages:
 - as(1)
 - cc(1)
 - cps(3)
 - cxdb(1)
 - cxoi(1)
 - cxperf(1)
 - f77(1)
 - ld(1)
 - pthread(3t)

Ordering documents

To order additional copies of this document or other documents listed in the “Associated documents” section, call 1-800-227-8164 between 6 a.m. and 5 p.m. PST.

To place an order from outside the United States, or if you cannot use the 1-800 number, call 415-857-5027.

Please have the order number (xxxxx-9xxxx) and the exact title of the document available when ordering.

1 Introduction

This chapter introduces the Hewlett-Packard Exemplar C and Fortran 77 compilers. These compilers are based on the standard Hewlett-Packard C and Fortran 77 compilers and are designed for creating applications for HP-UX 11.0 or higher.

The programming model

The Exemplar compilers implement a subset of the Exemplar programming model, which provides advanced parallelism. This model supports the following programming paradigms:

- Shared-memory
- Message-passing
- Shared-memory/message-passing

In the shared-memory paradigm, the compilers perform optimizations and—if requested—parallelization. Directives and pragmas allow you to further increase optimization opportunities.

In the message-passing paradigm, the programmer uses functions to explicitly spawn parallel processes, share data among processes, and coordinate their activities.

The shared-memory/message-passing paradigm allows you to combine the two paradigms, taking advantage of their respective strengths.

This book focuses on the compiler support for the shared-memory paradigm. See the *Exemplar Programming Guide* for information on programming efficiently using the shared-memory paradigm.

See the *HP MPI User's Guide* for information on using message passing.

Standard HP compiler information

This section discusses some of the standard HP compiler options that are referenced later in this book. However, this book is a supplement to the standard HP compiler documentation. See the cc(1) and f77(1) man pages for:

- Command-line options that are used most often
- Optimization options
- Input files information
- Diagnostics information
- Environment variables

See the section “Associated documents” on page xi for a list of additional documentation.

NOTE

The Exemplar compilers perform optimizations beyond those found in the standard HP compilers. See the *Exemplar Programming Guide* for more information.

+O0 (default)

Optimization level +O0 is the default optimization level. Your code compiles fastest at this level, but with little optimization. Code development and debugging should be done at this level.

At optimization level +O0, the optimizations in Table 1 are performed.

Table 1

Optimizations performed at +O0

Optimization	Description
Constant folding	Replaces an operation on constant operands with the result of the operation
Partial evaluation of test conditions	Determines, where possible, the truth value of a logical expression without evaluating all the operands (also known as short-circuiting)

+O1

The transformations performed at +O1 are local to small subsections of code and, therefore, are performed quickly and with little runtime storage required by the compiler. Use +O1 when some optimization is desired, but when compile-time performance is more important than runtime performance.

At optimization level +O1, the optimizations listed in Table 2 are performed.

Table 2

Optimizations performed at +O1

Optimization	Description
+O0 optimizations	See Table 1
Branch optimizations	Changes branch instructions into more efficient sequences
Dead code elimination	Removes code that is unreachable or is otherwise never executed
Instruction scheduling	Schedules instructions to take advantage of pipelining
More efficient use of registers	
Peephole optimizations	Replaces assembly language instruction sequences with faster sequences and removes redundant register loads and stores

+O2, -O

You can use either -O or +O2 to enable the +O2 optimizations.

Transformations at +O2 are performed over the scope of each procedure. If you use this optimization level, the compiler uses more memory than at +O1 and takes longer to process your program. Optimizing procedures of more than 1,000 lines at this level takes considerably longer than at +O1.

At optimization level +O2, the optimizations in Table 3 are performed.

Table 3 Optimizations performed at +O2

Optimization	Description
+O0 and +O1 optimizations	See Table 1 and Table 2
Global register allocation	Determines when and how long commonly used variables and expressions occupy a register
Strength reduction of induction variables	Removes linear functions of a loop counter and replaces each function with a variable that contains the value of that function
Strength reduction of constants	Replaces some multiplication instructions with addition instructions
Common subexpression elimination	Replaces subsequent instances of an expression with its result
Advanced constant folding and propagation (Simple constant folding is done at +O0)	Replaces an operation on constant operands with result of the operation (constant folding) and replaces variable references with a constant value previously assigned to that variable (constant propagation)
Loop-invariant code motion	Recognizes instructions inside a loop where the results never change and moves those instructions outside the loop
Store/copy optimization	Substitutes registers for memory locations
Unused definition elimination	Removes unused references to memory locations and register definitions
Software pipelining	Rearranges the order in which instructions execute in a loop to prevent processor stalls
Register reassociation	Reduces the cost of computing address expressions for array references by dedicating a register to track the value of the address expression
Loop unrolling (innermost loops)	Increases a loop's step value and replicates the loop body, with each replication appropriately offset from the induction variable so that all iterations are performed given the new step

+O3

At optimization level +O3, the following optimizations are made:

- The +O0, +O1, and +O2 optimizations (See Table 1, Table 2, and Table 3)
- Loop transformations such as distribution, interchange, fusion, vectorization, loop unrolling (non-innermost loops), loop blocking, loop unroll and jam (in C V2.0), and parallelization
- Hoisting conditional code out of loops (IF-DO interchange)
- Cloning within a file: creating a specialized version of a subprogram that can be optimized on a per-call-site basis
- Subprogram inlining within a file

+O4

At this level, optimization occurs at link time, allowing the optimizer to analyze all files compiled with the +O4 option at once. Because analysis is done when linking, the compile time is generally shorter than at lower optimization levels, but linking takes more time.

At optimization level +O4, the following optimizations are made:

- The +O0, +O1, +O2, and +O3 optimizations (See Table 1, Table 2, Table 3, and the section “+O3” above)
- Cloning across all files in the program that have been compiled at +O4
- Inlining across all files in the program that have been compiled at +O4

+O[no]aggressive

The `+O[no]aggressive` option enables optimizations that can result in significant performance improvement, but that can change a program's behavior. These optimizations include those invoked by the following advanced options (which are described in the `cc(1)` and `f77(1)` man pages):

- `+Osignedpointers` (available only in C)
- `+Oentrysched`
- `+Onofltacc`
- `+Olibcalls`
- `+Onoinitcheck`
- `+Ovectorize`

The default is `+Onoaggressive`. The `+O[no]aggressive` option can be used at `+O2` and above.

+O[no]all

The `+Oall` option applies maximum optimization to achieve the best runtime performance. This option is equivalent to specifying `+Oaggressive` and `+Onolimit` on the same command line. The `+Oall` option implies `+O4`. The default is `+Onoall`.

+O[no]fail_safe

The `+Ofail_safe` option allows a compilation with internal optimization errors to continue, rather than abort. If internal optimization errors are found, the compiler issues a warning message, then restarts the compilation at `+O0`. When using `+Onofail_safe`, compilation aborts if internal optimization errors occur.

This option can be used at `+O1` or higher. The default is `+Ofail_safe`.

+O[no]info

The `+O[no]info` option displays [does not display] feedback information about the optimization process (for example, cloning and inlining). Currently, this option is useful only at `+O3` and above. The default is `+Onoinfo`. For information on a related option, see the section “`+O[no]report[=report_type]`” on page 19.

+Oinline_budget=*n*

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 default is `+Oinline_budget=100`.

The `+Onolimit` and `+Osize` options also affect inlining. Specifying the `+Onolimit` option implies specifying `+Oinline_budget=200`. The `+Osize` option implies `+Oinline_budget=1`.

Note, however, that the `+Oinline_budget` option 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 command line.

The `+Oinline_budget=n` option is valid at `+O3` and above.

+O[no]limit

The `+O[no]limit` option suppresses [does not suppress] optimizations that significantly increase compile-time or consume large amounts of memory. Specifying `+Onolimit` implies specifying `+Oinline_budget=200`. (See the section “`+Oinline_budget=n`” above for additional information.) This option can be used at `+O2` and above. The default is `+Olimit`.

+O[no]loop_transform

The `+O[no]loop_transform` option transforms [does not transform] eligible loops for improved cache performance. The transforms include loop distribution, loop interchange, loop blocking, loop unroll, loop unroll and jam (in C V2.0), and loop fusion. This option can be used at `+O3` and above. The default is `+Oloop_transform`.

+O[no]loop_unroll[=*n*]

This option unrolls [does not unroll] program loops by a factor of *n*. For example, specifying `+Oloop_unroll=4` requests the optimizer to replicate the loop body four times. This option can be used at `+O2` and above. The default is `+Oloop_unroll=4`.

+O[no]size

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 7 for additional information.

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

The `+O[no]size` option can be used at `+O2`, `+O3`, or `+O4`. The default is `+Onosize`.

Compiler usage

The examples in this section demonstrate the use of the C and Fortran 77 compilers. The functionality and options illustrated in any example in the book apply to both the C and Fortran 77 compilers.

Using the C compiler

The two C compilers are described below:

- `cc` is the Exemplar C compiler and is located at `/opt/ansic/bin/cc`.
- `c89` is the Exemplar POSIX-conforming C compiler and is located at `/opt/ansic/bin/c89`.

The remainder of this book refers to the `cc` compiler. Any `cc` example also applies to the `c89` compiler.

The `cc` compiler command has the form:

```
% cc [options] files
```

where

options Is zero or more C compiler options

files Is a space-separated list of one or more files

For example, the following command

```
% cc prog1.c prog2.c prog3.c
```

compiles the three files (`prog1.c`, `prog2.c`, `prog3.c`) and produces an executable, which is named `a.out` by default.

In this command,

```
% cc -o prog prog.c proc1.o
```

`cc` compiles `prog.c` to produce the object file `prog.o`, then calls the linker `ld` to link `prog.o` and `proc1.o` with the default start-up routines and library routines. The file `prog.o` is deleted after linking. The `-o prog` causes the resulting executable file to be named `prog` instead of `a.out`.

For additional information, see the `cc(1)` man page.

Using the Fortran 77 compiler

The two Fortran 77 compilers are described below:

- `f77` is the Exemplar Fortran 77 compiler and is located at `/opt/fortran/bin/f77`.
- `fort77` is the Exemplar POSIX-conforming Fortran 77 compiler and is located at `/opt/fortran/bin/fort77`.

The remainder of this book refers to the `f77` compiler. Any `f77` example also applies to the `fort77` compiler.

The `f77` compiler command has the form:

```
% f77 [options] files
```

where

options Is zero or more Fortran 77 compiler options

files Is a space-separated list of one or more files

For example, the command

```
% f77 -c prog.f
```

compiles the file `prog.f` to produce the object file `prog.o`, then (because of the `-c` option) suppresses linking. The `prog.o` file can be linked later by including it on a `f77` command line or by using the linker (`ld`) directly.

In the following example,

```
% f77 -v prog1.f prog2.f
```

the verbose mode is enabled by using `-v`. When compiling in verbose mode, the compiler displays (to standard error) a step-by-step description of the compilation process.

This command

```
% f77 +O3 +Oparallel prog.f
```

shows the request of level 3 optimizations (`+O3`) and the request that the compiler honor the parallelism directives of the Exemplar programming model and generate parallel code where appropriate (`+Oparallel`). The `+Oparallel` option is only valid at `+O3` and above.

For additional information, see the `f77(1)` man page.

Options to get you started

This section highlights options that you may want to use regularly with the Exemplar compilers. The options are performance-related and are described only briefly in this section; however, sources for more information are included, where available.

Table 4 Options to get you started

Option	Description
+O3	Invoke level 3 optimizations. See the section “+O3” on page 5 for more information.
+O3 +Oparallel	Invoke level 3 optimizations and cause the compiler to honor parallelism directives and pragmas from the Exemplar programming model and to generate parallel code where appropriate. If you compile with +O3 +Oparallel, be sure to also link with +O3 +Oparallel (if you link separately). See the section “+O[no]parallel” on page 18 for more information.
+Odataprefetch	Prefetch data referenced in loops. See the section “+O[no]dataprefetch” on page 15 for more information.
+Ofltacc	Disable floating-point optimizations that can result in numerical differences. See the cc(1) or f77(1) man page for more information.
+Oinfo	Display information on the optimization process. See the section “+O[no]info” on page 7 for more information.
+Olibcalls	Use low-call-overhead versions of select library routines. See the cc(1) or the f77(1) man page for more information.
+Onoautopar	Request that the compiler parallelize only those loops with prefer_parallel or loop_parallel directives or pragmas. See the section “+O[no]autopar” on page 15 for more information.

Introduction
Options to get you started

Option	Description
+Onolimit	Do not suppress optimizations that significantly increase compile-time or consume large amounts of memory. See the section “+O[no]limit” on page 7 for more information.
+Onoparmsoverlap	Optimize with the assumption that subprogram arguments do not refer to the same memory. When this option can be used, it allows the compiler to generate significantly faster code. See the cc(1) or the f77(1) man page for more information.
+Oreport	Display optimization reports. See the section “+O[no]report[= <i>report_type</i>]” on page 19 for more information.
-Wl,-aarchive_shared	(For use when linking with the compiler driver) Search the archive version of a library; if the archive version is not available, search the shared version of the library.
-Wl,+FPD	(For use when linking with the compiler driver) Enable sudden underflow (flush to zero) of denormalized values.

*Assuming +Onoexemplar_model is not also specified (See the section “+O[no]exemplar_model” on page 16 for more information.)

2

Exemplar extensions

This chapter describes:

- Options that are specific to the Exemplar compilers and explains how they are used
- The available directives and pragmas
- Exemplar Fortran 77 language extensions and intrinsics
- Exemplar Fortran 77 equivalences
- Large files support and predefined C preprocessor symbols

Exemplar compilers recognize the options, directives, and pragmas that the standard HP compilers recognize. Extensions accepted by the Exemplar compilers, however, are not recognized by the standard HP compilers. The following sections describe these extensions. See Chapter 1, “Introduction,” for an overview of the standard HP compiler options discussed below.

Exemplar compiler options

The options below are recognized in addition to those supported by the standard HP compilers or are available in the standard HP compilers, but have been modified to behave differently in the Exemplar compilers.

-g

This option requests that the compiler generate debugging information in the executable file that can be used by the CXdb debugger (an optional product). See Chapter 5, “Debugging and profiling,” for more information on CXdb.

NOTE

Debugging with the dde and xdb debuggers is not supported with code compiled using the Exemplar compilers.

If `-g` is specified when compiling, the Exemplar C and Fortran 77 compilers restrict optimizations to the `+O0` level.

-I8

This option specifies that `INTEGER` and `LOGICAL` variable declarations with unspecified lengths are to occupy 8 bytes of storage.

Also, this option transforms intrinsic function references that return default integer or logical values to return 8-byte values of the specified type.

+O[no]autopar

When used with the `+Oparallel` option, `+Oautopar` (the default) causes the compiler to automatically parallelize loops that are safe to parallelize.

A loop is safe to parallelize if it has an iteration count that can be determined at runtime before loop invocation, and contains 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.

You can use Fortran directives and C pragmas to improve on the automatic optimizations and to assist the compiler in locating additional opportunities for parallelization.

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.

Because parallelization takes place only at `+O3` and above, `+O[no]autopar` is useful only at `+O3` and above.

+O[no]dataprefetch

The `+O[no]dataprefetch` option enables [disables] optimizations to generate data prefetch instructions for data referenced within innermost loops. The effect is that the memory system will retrieve the data for future iterations while the processor is executing current iterations. For cache lines containing data that will be written, `+Odataprefetch` prefetches the cache lines so that they are valid for both read and write access.

This option provides no benefit to loops whose data fits in the cache; in fact, it can slow them down because of the prefetch instructions. For loops whose data does not fit in the cache, the speedup can be substantial.

The `+O[no]dataprefetch` option is valid at `+O2` and above. The default is `+Onodataprefetch`.

+O[no]dynsel

When specified with `+Oparallel`, `+Odynsel` (the default) 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 can be used to enable dynamic selection for specific loops when `+Onodynsel` is in effect.

+O[no]exemplar_model

`+Oexemplar_model` (the default) causes the compiler to recognize the Exemplar programming model. This option allows you to use the directives, pragmas, and associated command-line options that make up the programming model. At lower optimization levels (`+O0`, `+O1`, `+O2`), this option enables only the following components of the programming model:

- Synchronization directives (Fortran)
- Synchronization pragmas and synchronization typedefs (C)
- Memory class directives (Fortran)
- Memory storage class specifiers (C)

At `+O3` and `+O4`, using `+Oexemplar_model` enables all directives, pragmas, storage class specifiers, and typedefs. See the section “Exemplar compiler directives and pragmas” on page 22 for additional information.

The `+Onoexemplar_model` option turns off support for the Exemplar programming model. If you use this option, directives and pragmas from the Exemplar programming model are ignored.

This option is available only in C V1.2.3 and Fortran 77 V1.2.3.
In C V2.0, `+Oexemplar_model` is on by default, and
`+Onoexemplar_model` is not available.

+O[no]loop_block

Optimization level(s): +O3, +O4

Default: `+Onoloop_block`

The `+O[no]loop_block` option (available only in C V2.0) enables [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 *Exemplar Programming Guide*.

+O[no]loop_unroll_jam

Optimization level(s): +O3, +O4

Default: `+Oloop_unroll_jam`

The `+O[no]loop_unroll_jam` option (available only in C V2.0) enables [disables] loop unrolling and jamming. The `+Onoloop_unroll_jam` option 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 *Exemplar Programming Guide*.

+O[no]parallel

The +Oparallel option causes the compiler to:

- Honor the directives and pragmas of the Exemplar programming model that involve parallelism, such as `begin_tasks`, `loop_parallel`, `prefer_parallel`, and `parallel`. These directives and pragmas are not recognized if +Onoexemplar_model is specified.
- Look for opportunities for parallel execution in loops.

The following methods can be used to specify the number of processors used in your parallel program:

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

For more information on these directives and pragmas see the section “Exemplar compiler directives and pragmas” on page 22.

- The environment variable `MP_NUMBER_OF_THREADS`, 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. If `MP_NUMBER_OF_THREADS` is not set, your program runs on the number of processors in the system where it is executing.

The +Oparallel option is valid only at optimization level +O3 and above. Using the +Oparallel option disables +Ofail_safe, which is on by default. See the section “+O[no]fail_safe” on page 6 for more information.

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

NOTE

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

+O[no]report [=report_type]

This option causes the compiler to display various optimization reports. `+Onoreport` is the default. The value of *report_type* determines which report is displayed, as described below.

`+Oreport=loop` produces the Loop Report. This report gives information on optimizations performed on loops and calls. Using `+Oreport` (without *report_type*) also produces the Loop Report.

`+Oreport=private` produces the Loop Report and the Privatization Table, which provides information on loop variables that are privatized by the compiler.

`+Oreport=all` produces all reports.

The `+Oreport [=report_type]` option is active only at `+O3` and above. The `+Onoreport` option does not accept any of the *report_type* values. See the *Exemplar Programming Guide* for more information on the optimization reports.

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

+O[no]sharedgra

The `+Onosharedgra` option disables global register allocation for shared-memory variables that are visible to multiple threads. This option can help if a variable shared among parallel threads is causing wrong answers. See the *Exemplar Programming Guide* for more information.

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

+pa

The `+pa` option requests that the application be compiled for routine-level profiling with CXperf. The `+pa` option is not valid with the `+O4` or `+Oall` optimization levels. Also, `+pa` is not compatible with the `-p` or `-G` options. See Chapter 5, “Debugging and profiling,” for more information on CXperf.

+pal

At `+O2` and `+O3`, the `+pal` option requests that the application be compiled for routine-level and loop-level profiling with CXperf. The `+pal` option is not valid with the `+O4` or `+Oall` optimization levels. Also, `+pal` is not compatible with the `-p` or `-G` options. See Chapter 5, “Debugging and profiling,” for more information on CXperf.

+tm *target*

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:

- `spp1200` to specify SPP1200 Series machines
- `spp1600` to specify SPP1600 Series machines
- `S2000` to specify S2000 servers
- `X2000` to specify X2000 servers

In addition to the values above, the Exemplar C V2.0 compiler accepts the following values for *target*:

- `K7200` to specify K-Class servers using PA-7200 processors
- `K8000` to specify K-Class servers using PA-8000 processors
- `V2200` to specify V2200 servers

Although Fortran 77 is available on the K-Class and V-Class servers, these *target* values are not available.

This option is valid at all optimization levels. The default *target* value corresponds to the machine on which you invoke the compiler. The `+tm target` option is automatically specified when you use one of the Exemplar compiler drivers.

Using the `+tm target` option implies `+DA` and `+DS` settings as described in Table 5. `+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 `cc(1)` man page or the `f77(1)` man page for more information on the `+DA` and `+DS` options.

Table 5 `+tm target` and `+DA/+DS`

<i>target</i> value specified	<code>+DAarchitecture</code> implied	<code>+DSmodel</code> implied
spp1200	1.1	1.1
spp1600	1.1	1.1
S2000	2.0	2.0
X2000	2.0	2.0
K7200	1.1	1.1
K8000	2.0	2.0
V2200	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`.

Exemplar compiler directives and pragmas

This section presents an alphabetical list of the Fortran directives and C pragmas that make up the Exemplar programming model. The Exemplar compilers accept the directives and pragmas listed below in addition to those supported by the standard HP compilers.

This section is intended to provide a brief overview of the available directives and pragmas. More specific information and examples can be found in the *Exemplar Programming Guide*. The Fortran directives not supported as C pragmas are expressed in C as either storage class extensions (`thread_private`, etc.) or as typedefs (`gate_t`, `barrier_t`, etc.) in the `spp_prog_model.h` file and are described in the “Memory classes” and the “Advanced shared-memory programming” chapters of the *Exemplar Programming Guide*.

The form of an Exemplar Fortran compiler directive is:

```
C$DIR directive-list
```

The form of an Exemplar C pragma is:

```
#pragma _CNX directive-list
```

where

directive-list Is a comma-separated list of the directives/pragmas described in this chapter.

For information on how to properly use these directives or pragmas, see the *Exemplar Programming Guide*.

Directive names are presented here in lowercase; they may be specified in either case in both languages, but `#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.

align_cti(*namelist*)

This directive or pragma aligns the variables and arrays listed in *namelist* on CTIcache boundaries. This allows for more efficient data reuse.

A CTIcache is a partition of physical memory that exists on each hypernode and is used to store copies of global data fetched from other hypernodes. (A hypernode is a set of processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system microkernel.)

Single-hypernode systems do not have CTIcaches; however, this directive can be useful if you are porting code to a multi-hypernode system. See the *Exemplar Programming Guide* for more information.

barrier(*namelist*)

This Fortran directive denotes a list of variables, as given in *namelist*, that are to be used as the synchronization variables for the barrier routines. This does not imply any synchronization in itself; it is simply defining the barrier variables. In C, `barrier` is a typedef (`barrier_t`), rather than a pragma. For more information, refer to the *Exemplar Programming Guide*.

begin_tasks[(*attribute_list*)]

This directive or pragma defines the beginning of a section (or sections; see `next_task`) of code that is to be executed as an independent, parallel task. Each task is executed by a separate thread. `begin_tasks` must have an accompanying `end_tasks` in the same program unit.

The optional *attribute_list* can be any of the following legal combinations (*m* is an integer constant):

- threads (default)
- nodes
- dist
- ordered
- max_threads=*m*
- threads, ordered
- nodes, ordered
- dist, ordered
- threads, max_threads=*m*
- nodes, max_threads=*m*
- dist, max_threads=*m*
- ordered, max_threads=*m*
- threads, ordered, max_threads=*m*
- nodes, ordered, max_threads=*m*
- dist, ordered, max_threads=*m*

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to the *Exemplar Programming Guide* for a complete discussion of parallel tasking.

block_loop[(block_factor=*n*)]

This directive or pragma indicates a specific loop to block, and optionally, the block factor *n* (*n* must be an integer constant greater than or equal to 2) that is to be used in the compiler's internal computation of loop nest based data reuse. If no `block_factor` is specified, the compiler uses a heuristic to determine the `block_factor`. Refer to the *Exemplar Programming Guide* for more information on blocking.

block_shared(*allocatable_array_namelist*)

This Fortran directive is used to declare arrays as being of type `block_shared`. Block-shared arrays are sized to be an integral multiple of the page size. The pages of the array are distributed in same-size blocks across the hypernodes on which the process is executing in the system. If the user-specified size is not an integral multiple of `page_size * num_nodes()`, then the size is automatically rounded up to meet this criterion. Refer to Chapter 5, "Memory classes," in the *Exemplar Programming Guide* for more information.

critical_section[(*gate_var*)]

This directive or pragma defines the beginning of a code block in which only one thread may be executing at a time. The end of the code block must be indicated by an `end_critical_section` directive or pragma, which must appear in the same flow of control within the same program unit. The optional *gate_var* can be used to differentiate between parallel tasks. Refer to the *Exemplar Programming Guide* for more information.

dynsel[(*trip_count=n*)]

This directive or pragma enables workload-based dynamic selection for the immediately following loop. *trip_count* represents either the `thread_trip_count` or `node_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. When `node_trip_count=n` is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the node-parallel version is run, assuming `+Onodepar` is specified.

[Exemplar extensions](#)

[Exemplar compiler directives and pragmas](#)

end_critical_section

This directive or pragma defines the end of the critical section that was begun with the `critical_section` directive or pragma. `critical_section` and `end_critical_section` must appear as a pair. Refer to the *Exemplar Programming Guide* for more information.

end_ordered_section

This directive or pragma defines the end of the ordered section that was begun with the `ordered_section` directive or pragma. `ordered_section` and `end_ordered_section` must appear as a pair. Refer to the *Exemplar Programming Guide* for more information on ordered sections.

end_parallel

This directive or pragma signifies the end of a parallel region. The `parallel` directive signifies the beginning of a parallel region. Refer to Chapter 4, “Basic shared-memory programming,” in the *Exemplar Programming Guide* for more information.

end_tasks

This directive or pragma terminates the specification of parallel tasks indicated by `begin_tasks` and `next_task`. It must appear at the end of the last section of parallel code defined by these directives or pragmas. All of these must appear in the same program unit. Refer to the *Exemplar Programming Guide* for more information.

far_shared(*namelist*)

This Fortran directive causes the compiler to place the data objects in *namelist* (variables, arrays, or COMMON blocks) into `far_shared` memory. `far_shared` memory is the most general form that is distributed on a page basis across the memories of all hypernodes in a system. The `far_shared` data objects of a process are addressable by all threads of that process. In C, `far_shared` is a storage class specifier. Refer to the *Exemplar Programming Guide* for more information on memory classes.

far_shared_pointer(*namelist*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in *far_shared* memory, regardless of the memory classes to which the respective objects are allocated.

This directive applies only to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” in the *Exemplar Programming Guide* for more information on memory classes.

gate(*namelist*)

This Fortran directive defines a gate variable that is to be used subsequently in a critical section, ordered section, or passed as an argument to the synchronization intrinsics. In C, *gate* is a typedef (*gate_t*), rather than a pragma. Refer to the *Exemplar Programming Guide* for more information.

loop_parallel[(*attribute_list*)]

This directive or pragma is an explicit instruction to the compiler to parallelize the immediately following loop. The loop iterations are run in an indeterminate order unless the optional *ordered* attribute appears. You are responsible for any required data privatization and loop synchronization, as described in Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” of the *Exemplar Programming Guide*. The optional *attribute_list* can be any of the following combinations (*n* and *m* are integer constants):

- threads (default)
- nodes
- dist
- ordered
- max_threads=*m*
- chunk_size=*n*
- threads, ordered

Exemplar extensions

Exemplar compiler directives and pragmas

- `nodes, ordered`
- `dist, ordered`
- `threads, max_threads=m`
- `nodes, max_threads=m`
- `dist, max_threads=m`
- `ordered, max_threads=m`
- `threads, chunk_size=n`
- `nodes, chunk_size=n`
- `dist, chunk_size=n`
- `threads, ordered, max_threads=m`
- `nodes, ordered, max_threads=m`
- `dist, ordered, max_threads=m`
- `chunk_size=n, max_threads=m`
- `threads, chunk_size=n, max_threads=m`
- `nodes, chunk_size=n, max_threads=m`
- `dist, chunk_size=n, max_threads=m`
- `ivar= indvar`

The `ivar= indvar` attribute is:

- **Required for all loops in C and for DO WHILE and hand-rolled loops in Fortran**
- **Optional for Fortran DO loops**
- **Compatible with any other attribute**

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to the *Exemplar Programming Guide* for more information.

loop_private(*namelist*)

This directive or pragma declares a list of variables and/or arrays private to the immediately following loop. No values may be carried into the loop by `loop_private` variables. To be loop private, the variables and/or arrays must be assigned before they are used on each iteration of the immediately following loop. These private data items are distinct from the shared items of the same name that exist outside the loop. Values assigned to `loop_private` variables on the final iteration (that is, the n th iteration of a loop with n iterations) may be saved into the shared variables of the same name if the `save_last` directive or pragma also appears on this loop. If `save_last` is not used, then the value of any shared variable declared to be `loop_private` is undefined at loop termination. Refer to the *Exemplar Programming Guide* for more information.

near_shared(*namelist*)

When applied to static variables at compile-time, this Fortran directive causes all pages of the data objects in *namelist* to be mapped to physical pages on logical hypernode 0 (the hypernode where the program starts). If applied to allocatable arrays, then the pages of such arrays will be mapped to physical pages on the hypernode of the allocating thread. `near_shared` data can be addressed by any thread of a process on any hypernode in the system but it is “closer” (in terms of access latency) to the threads on the hypernode that allocates the data. In C, `near_shared` is a storage class specifier. Refer to the *Exemplar Programming Guide* for more information on memory classes.

near_shared_pointer(*namelist*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `near_shared` memory, regardless of the memory classes to which the respective objects are allocated.

This directive applies only to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” in the *Exemplar Programming Guide* for more information on memory classes.

next_task

This directive or pragma starts a block of code following a `begin_tasks` block that will be executed as a parallel task. The end of the code block is marked by another `next_task` or by an `end_tasks` directive or pragma.

This directive must appear within a `begin_tasks` and `end_tasks` pair. There is no limit on the number of `next_task` directives that can appear. Refer to the *Exemplar Programming Guide* for more information.

no_block_loop

This directive or pragma disables loop blocking on the immediately following loop. Refer to the *Exemplar Programming Guide* for more information on loop blocking.

no_distribute

This directive or pragma disables loop distribution for the immediately following loop. Refer to the *Exemplar Programming Guide* for more information on loop distribution.

no_dynsel

This directive or pragma disables workload-based dynamic selection for the immediately following loop. Refer to the *Exemplar Programming Guide* for more information on dynamic selection.

no_loop_dependence (*namelist*)

This directive or pragma 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 the possibility of parallelization.

Refer to the *Exemplar Programming Guide* for more information.

no_loop_transform

This directive or pragma 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 or pragma appears. Refer to the *Exemplar Programming Guide* for more information.

no_parallel

This directive or pragma prevents the compiler from generating parallel code for the immediately following loop. Refer to the *Exemplar Programming Guide* for more information.

no_side_effects(*funclist*)

This directive or pragma 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; also, the compiler may be able to parallelize loops with calls when informed that the called routines do not have side effects.

no_unroll_and_jam

This directive or pragma disables loop unroll and jam for the immediately following loop. Refer to the *Exemplar Programming Guide* for more information.

node_private(*namelist*)

This Fortran directive causes the variables and arrays specified in *namelist* to be replicated in the physical memory of each hypernode on which the process is executing. Thus, while each data object has a single image in virtual memory, it maps to a different physical location on each hypernode. The threads of a process within a hypernode all share access to the copy on their hypernode and cannot access the copies on other hypernodes. In C, `node_private` is a storage class specifier. Refer to Chapter 5, “Memory classes,” in the *Exemplar Programming Guide* for more information.

node_private_pointer(*namelist*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `node_private` memory, regardless of the memory classes to which the respective objects are allocated.

This directive applies only to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” in the *Exemplar Programming Guide* for more information.

ordered_section(*gate_var*)

This directive or pragma defines the beginning of an ordered section. An ordered section is the same as a critical section (a code block in which only one thread may be executing at a time) with the additional restriction that the threads must pass through the ordered section in iteration order. The end of the code block must be indicated by an `end_ordered_section` directive or pragma. Ordered sections must appear within the control flow of a `loop_parallel(ordered)` directive. Refer to the *Exemplar Programming Guide* for more information.

parallel[(*attribute_list*)]

This directive or pragma signifies the beginning of a parallel region of code. All code up to the following `end_parallel` directive or pragma will be run on all available threads. No loop transformations, data privatization, or parallelization analysis will be performed by the compiler on the code in the region.

The optional *attribute_list* can be any of the following legal combinations (*m* is an integer constant):

- `threads` (default)
- `nodes`
- `max_threads=m`
- `threads,max_threads=m`
- `nodes,max_threads=m`

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to Chapter 4, “Basic shared-memory programming,” in the *Exemplar Programming Guide* for more information.

parallel_private(*namelist*)

This directive or pragma declares a list of variables or arrays private to the immediately following parallel region. It serves the same purpose for parallel regions that `task_private` serves for tasks. The privatized variables and arrays will not carry their values beyond the `end_parallel` directive or pragma. Refer to Chapter 4, “Basic shared-memory programming,” in the *Exemplar Programming Guide* for more information.

prefer_parallel [(*attribute_list*)]

This directive or pragma instructs the compiler to parallelize the following loop, but only if it is safe to do so. A loop is safe to parallelize if it has an iteration count that can be determined at runtime before loop invocation and contains 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.) Refer to the *Exemplar Programming Guide* for more information.

The optional *attribute_list* can be any of the following combinations (*n* and *m* are integer constants):

- threads (default)
- nodes
- dist
- max_threads=*m*
- chunk_size=*n*
- threads, max_threads=*m*
- nodes, max_threads=*m*
- dist, max_threads=*m*
- threads, chunk_size=*n*
- nodes, chunk_size=*n*
- dist, chunk_size=*n*
- chunk_size=*n*, max_threads=*m*
- threads, chunk_size=*n*, max_threads=*m*
- nodes, chunk_size=*n*, max_threads=*m*
- dist, chunk_size=*n*, max_threads=*m*

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

reduction(*namelist*)

This pragma (available in CV2.0)—which is only to be used with `loop_parallel`—specifies that the scalar variables in the comma-separated *namelist* are involved in reductions. The `reduction` pragma is used to inform the compiler of reductions in `loop_parallel` loops. Once the compiler is informed of the reductions, the compiler generates code to perform the reduction while parallelizing the loop—assuming no other parallelization inhibitors occur in the loop. Refer to the *Exemplar Programming Guide* for more information.

save_last[(*list*)]

This directive or pragma specifies that the variables in the comma-separated *list* that are also named in an associated `loop_private(namelist)` directive or pragma must have their last values saved into the “shared” variable of the same name at loop termination. (A variable’s last value in a loop of *n* iterations is the value it is assigned in the *n*th iteration.)

If the optional *list* is not used, `save_last` specifies that all variables named in an associated `loop_private(namelist)` directive or pragma must have their last values saved into the “shared” variable of the same name at loop termination.

If `save_last` is not specified then the values in any privatized variables or arrays are indeterminate at loop termination. Refer to the *Exemplar Programming Guide* for more information.

scalar

This directive or pragma 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 or pragma appears.

The `no_loop_transform` directive or pragma provides the same functionality as the `scalar` directive or pragma and is recommended in place of the `scalar` directive or pragma.

sync_routine(*routinelist*)

This directive or pragma indicates to the compiler that the routines listed in *routinelist* are user-defined synchronization routines, so that the compiler does not attempt to move code across these routine calls. Use `sync_routine` anytime you hide a call to a compiler synchronization function inside another routine call, or anytime you use CPSlib functions for synchronization. (CPSlib is a library of low-level parallelization and synchronization routines. See the *Exemplar Programming Guide* for more information.)

`sync_routine` is effective only for the listed routines in the file in which it appears.

task_private(*namelist*)

This directive or pragma privatizes the variables and arrays specified in *namelist* for each task specified in the immediately following `begin_tasks/end_tasks` block. If a `task_private` data object is referenced within a task, it must have been assigned a value previously in that task. The privatized variables and arrays do not carry their values beyond the `end_tasks` directive or pragma. Refer to the *Exemplar Programming Guide* for more information.

thread_private(*namelist*)

This Fortran directive causes the variables and arrays specified in *namelist* to be treated as being `thread_private`. `thread_private` data objects map to unique `node_private` addresses for each thread of a process. In C, `thread_private` is a storage class specifier. Refer to the *Exemplar Programming Guide* for more information.

thread_private_pointer(*namelist*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `thread_private` memory, regardless of the memory classes to which the respective objects are allocated.

This directive applies only to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” in the *Exemplar Programming Guide* for more information.

unroll_and_jam[(unroll_factor=*n*)]

This directive or pragma 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. Refer to the *Exemplar Programming Guide* for more information.

Exemplar Fortran 77 language extensions

This section describes the extensions that are supported in the Exemplar Fortran 77 compiler. See the *HP FORTRAN/9000 Programmer's Reference* for information on the extensions available in the standard HP Fortran 77 compiler.

INTEGER*8

The `INTEGER*8` data type allocates storage for 8-byte integer data.

INTEGER*8 constants

You can specify an `INTEGER*8` constant by adding the `K` suffix after the constant value. Using the `K` suffix is the only way to specify an `INTEGER*8` constant; the command-line option `-I8` does not imply `INTEGER*8` constants.

LOGICAL*8

The `LOGICAL*8` data type allocates storage for 8-byte logical data.

[Exemplar extensions](#)

[Exemplar Fortran 77 language extensions](#)

TASK COMMON

Exemplar Fortran supports Cray `TASK COMMON` blocks. A program should already be running multiple threads before calling a subroutine that contains a `TASK COMMON` block.

Variables in a `TASK COMMON` block are stored in a thread-private `COMMON` block (each thread has its own thread-local copy of the `TASK COMMON` block).

The `TASK COMMON` statement creates these blocks and has the form:

```
TASK COMMON /cbn/nlist[ , /cbn/nlist] . . .
```

where

<i>cbn</i>	Is a symbolic name for a <code>TASK COMMON</code> block. Unnamed <code>TASK COMMON</code> blocks are not allowed.
<i>nlist</i>	Is a list of variable names, array names, and array declarators. These variables cannot appear in a <code>DATA</code> statement, but otherwise can be used like any variables in <code>COMMON</code> storage.

All occurrences of the `TASK COMMON` block must be declared `TASK COMMON`; a `COMMON` block cannot be declared both `COMMON` and `TASK COMMON`. `TASK COMMON` blocks can be declared only in functions, subprograms and `BLOCK DATA` subprograms.

Using `TASK COMMON` is the same as using a `COMMON` block that is specified in the *namelist* of a `thread_private(namelist)` directive.

Exemplar Fortran 77 intrinsics

Table 6 describes the intrinsics in Exemplar Fortran 77 that support
INTEGER*8 data.

Table 6 **Intrinsic functions**

Entry point	Description	Specific intrinsic
BTEST_8	Bit test of an integer value	LOGICAL(8) function BKTEST(I,POS) INTEGER(8) :: I,POS
FTN_KQNINT	Nearest integer	INTEGER(8) function KIQNNT(A) REAL(16) :: A
FTN_KSIGN	Absolute value of A times B	INTEGER(8) function KISIGN(A,B) INTEGER(8) :: A,B
FTN_KZEXT_B1	Zero extend	INTEGER(8) function KZEXT(A) LOGICAL(8) :: A
IBCLR_8	Clear a bit to zero	INTEGER(8) function KIBCLR(I,POS) INTEGER(8) :: I,POS
IBITS_8	Extract a sequence of bits	INTEGER(8) function KIBITS(I,POS,LEN) INTEGER(8) :: I,POS,LEN
IBSET_8	Set a bit to one	INTEGER(8) function KIBSET(I,POS) INTEGER(8) :: I,POS
ISHFT_8	Logical shift	INTEGER(8) function KISHFT(I,SHIFT) INTEGER(8) :: I,SHIFT
ISHFTC_8	Circular shift of rightmost bits	INTEGER(8) function KISHFTC(I,SHIFT,SIZE) INTEGER(8) :: I,SHIFT INTEGER(8),OPTIONAL :: SIZE
KABS	Integer absolute value	INTEGER(8) function KIABS(A) INTEGER(8) :: A
KDIM	Positive difference	INTEGER(8) function KIDIM(X,Y) INTEGER(8) :: X,Y
KIDNINT	Nearest integer	INTEGER(8) function KIDNNT(A) DOUBLE PRECISION :: A

Exemplar extensions
Exemplar Fortran 77 equivalences

Entry point	Description	Specific intrinsic
KININT	Nearest integer	INTEGER(8) function KNINT(A) REAL :: A
KMOD	Remainder function	INTEGER(8) function KMOD(A,P) INTEGER(8) :: A,P
MVBITS_8	Copy a sequence of bits from one data object to another	subroutine KMVBITS(FROM, FROMPOS, LEN, TO, TOPOS) INTEGER(8) :: FROM, TO INTEGER :: FROMPOS, TOPOS, LEN

Exemplar Fortran 77 equivalences

Fortran 77's EQUIVALENCE statement allows you to associate variables so that they share the same storage space. In the standard HP Fortran 77 compiler, equivalences are placed in static storage. In the Exemplar Fortran 77 compiler, however, equivalences are stored on the stack because the `-Wc, -local_equivs` option is used by default.

Predefined symbols

The items listed in this section are predefined and have special meanings.

NOTE

"__" indicates two adjacent underscore characters. There is no space between these characters. If a space is added, the compiler does not recognize the variable as a predefined symbol.

`__HP_CXD_SPP=1`

This symbol (which has two leading underscores) is always defined when using the Exemplar compilers. The preprocessor (`cpp`) predefines this symbol so that code can be conditionalized based on whether a file is being compiled using the Exemplar compilers.

`_REENTRANT=1`

This symbol (which has one leading underscore) is predefined for use by the include files. When it is predefined, reentrant versions of `libc` routines are called. When `_REENTRANT` is not predefined, some `libc` routines that are not reentrant are called. Calling a non-reentrant routine from within a parallel region is an error.

The compiler predefines this symbol when `+Oparallel` is specified with either `+O3` or `+O4`.

Exemplar extensions
Predefined symbols

3

Developing parallel applications

This chapter discusses the various methods available for producing parallel applications on HP-UX 11.0 systems. The topics covered in this chapter include:

- Compiler-generated parallelism
- HP MPI
- Pthreads
- The Compiler Parallel Support library (CPSlib)

Compiler-generated parallelism

The Exemplar Fortran 77 and Exemplar C compilers can automatically produce parallel code. With the insertion of source code directives or pragmas, you can further the amount of code the compilers are able to parallelize.

Using the `+Oparallel` compiler option

Using the `+Oparallel` option at `+O3` and above allows the compiler to automatically parallelize loops that are profitable to parallelize. Also, because `+Oexemplar_model` is on by default, the compiler recognizes the parallelism-related directives and pragmas of the Exemplar programming model.

The Exemplar compilers find parallelism at the loop level and generate parallel code that will automatically run on as many processors as are available at runtime. Normally, these are all the processors of the system on which your program is running—unless you specify a smaller number of processors.

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.

For more information on using the `+Oparallel` option, refer to the section “`+O[no]parallel`” on page 18 or to the *Exemplar Programming Guide*.

HP MPI

A Hewlett-Packard implementation of the MPI message-passing library, known as HP MPI, has been developed specifically for HP-UX systems. Special compilation utilities are provided with HP MPI for compiling applications written in Fortran 77 and C.

HP MPI is an optional product. It is a native implementation of version 1.2 of the Message-Passing Interface (MPI) standard. HP MPI is optimized for HP-UX workstations, servers, and Exemplar systems. It allows you to create and run applications composed of one or more processes that interact using the MPI communication model.

The default location for HP MPI is the `/opt/mpi` directory. Example programs are available in the `/opt/mpi/help` directory. For information on specific utilities, refer to the man page for the utility in question. Man pages are stored in the directory `/opt/mpi/share/man`. For more information on using HP MPI, refer to the *HP MPI User's Guide* (B6011-90001).

Pthreads

Pthreads (POSIX threads) refers to the Pthreads library of thread-management routines. For information on Pthreads routines, see the `pthread(3t)` man page.

Accessing Pthreads

When you use Pthreads routines, your program must include the `<pthread.h>` header file and the Pthreads library must be explicitly linked in to your program. For example, assume the program `prog.c` contains calls to Pthreads routines. To compile the program properly linking in the Pthreads library, use the command:

```
% cc -D_POSIX_C_SOURCE=199506L prog.c -lpthread
```

The `-D_POSIX_C_SOURCE=199506L` string specifies the appropriate POSIX revision level. In this case, the level is `199506L`.

The Compiler Parallel Support library (CPSlib)

CPSlib is the Compiler Parallel Support library—a library of low-level parallelization and synchronization routines. For information on CPSlib routines, see the `cps(3)` man page or the *Exemplar Programming Guide*.

Accessing CPSlib

In the Exemplar compilers, CPSlib is automatically linked in at +O3 (and above) when +Oparallel is specified.

If your program explicitly calls CPSlib routines or calls other libraries that use CPS routines and you are not linking at +O3 (or +O4) with +Oparallel, you must explicitly link in CPSlib as shown in the following example. Assume `prog.c` contains calls to CPSlib routines:

```
% cc prog.c -lpthread -lcps -lpthread
```

Linking in CPSlib requires specifying—in the order given—all of the string `-lpthread -lcps -lpthread`.

Developing parallel applications
The Compiler Parallel Support library (CPSlib)

4

The Exemplar assembler and linker

This chapter discusses some of the differences between the Exemplar versions of the assembler (`as`) and linker (`ld`) and the standard HP-UX versions on which they are based. Also, this chapter presents some examples of how to use the assembler and linker. See the following for more information:

- *Assembly Language Reference Manual* (assembler information)
- *Programming on HP-UX* (linker information)
- `as(1)` and `ld(1)` man pages

NOTE

You do not have to invoke the assembler or linker directly; the compiler drivers invoke them for you.

The assembler

An assembler is a program that converts assembly language programs into an object file suitable for processing by the linker `ld`.

The Exemplar HP-UX assembler `as` differs from the standard HP-UX `as` in only one way: it supports the `.parallel` directive. This directive is provided so that assembly code that was assembled using `/usr/convex/bin/as` will assemble under the Exemplar HP-UX assembler. The Exemplar assembler is the default assembler when using an Exemplar compiler.

Assembler usage

Assembler commands have the following form:

```
% as [options] [files]
```

where

options Is zero or more of the allowed assembler options (see the as(1) man page for information on options)

files Is a space-separated list of zero or more files containing assembly code; if no files are given upon invoking as, source text is read from standard input

NOTE

The .s files created by compiling with -S are intended to provide insight into how the compiler is processing your code. These files may not be suitable as input to the assembler.

The first example illustrates how to produce an object file from an assembly-language file named prog.s:

```
% ls
prog.s
% as prog.s
% ls
prog.o prog.s
```

The object file (prog.o) is now ready to be processed by the linker to produce an executable file:

```
% ld prog.o
% ls
a.out prog.o prog.s
```

By default, the executable is named a.out; the -o *filename* option to ld can be used to specify a different name (*filename*) for the executable.

The linker

A linker is a program that combines separate object files into a single object file or executable program.

The Exemplar linker, `ld`, is the default linker when compiling with an Exemplar compiler. It differs from the standard HP's `ld` in that it supports the memory classes of the Exemplar programming model. Also, the Exemplar linker is required for using the CXdb debugger and the CXperf profiler.

You can specify linker options on the compiler command line by using the `-w1` option. See the `ld(1)` man page and the `cc(1)` man page or the `f77(1)` man page for more information.

Object file formats: SOM and ELF

Two kinds of object files exist on HP-UX platforms:

- Standard Object Module (SOM) — for 32-bit applications
- Executable and Linking Format (ELF) — for 64-bit applications

The two formats cannot be mixed: all the object files in an application must be in either the SOM format or the ELF format.

Linking to debug or profile

The Exemplar linker, `ld`, is required if you want to use the CXdb debugger (`cxdb`) or the CXperf profiler (`cxperf`). The linker provides support that is needed by both these development tools.

Debugging information is generated by using the `-g` option to the compiler. Profiling information is generated by using the `+pa` option or the `+pal` option. See Chapter 5, “Debugging and profiling,” for more information on using these tools.

[The Exemplar assembler and linker](#)
[The linker](#)

Linker usage

See the `ld(1)` man page for the form of linker commands.

The preferred method for accessing the linker is through the C or Fortran 77 compiler driver. If you use a compiler driver, your program is linked with the proper libraries in the right order.

The first example below shows how to combine multiple object files into a single executable file; the executable file is named `a.out` by default:

```
% ls
file1.o    file2.o    file3.o
% ld file1.o file2.o file3.o
% ls
a.out      file1.o    file2.o    file3.o
```

In the example below, `cc` compiles `main.c` to produce the object file `main.o`. The compiler then passes control to the linker, which combines `main.o`, `sub1.o`, and `sub2.o` to produce an executable file. The file `main.o` is deleted following a successful link. Because `-o main` is specified, the executable file is named `main`.

```
% cc -o main main.c sub1.o sub2.o
% ls
main      main.c    sub1.o    sub2.o
```

5

Debugging and profiling

This chapter provides an overview of the debugging and performance analysis tools available on Exemplar systems. The programs discussed in this chapter are optional products. If you are unsure whether a product is installed on your system, check with your system administrator.

See the following documents for more information on these tools:

- *CXdb Quick Reference*
- CXdb online help system
- `cxdb(1)` man page
- CXperf online help system
- `cxperf(1)` man page
- `cxoi(1)` man page

NOTE

Debugging with the `dde` and `xdb` debuggers is not supported with code compiled using the Exemplar compilers.

The CXdb debugger

CXdb is a window-based, symbolic debugger that lets you debug Fortran, C, and C++ programs compiled with the Fortran 77, Fortran 90, C, or aC++ compilers on Exemplar systems.

To debug using CXdb, you must:

- Compile your code using the `-g` option to produce debugging information in the executable file for CXdb to read
- Link the application with the `-g` option using the Exemplar linker by means of the compiler driver
- Statically link the application with archived libraries

NOTE

If `-g` is specified when compiling, the Exemplar C and Fortran 77 compilers restrict optimizations to the `+O0` level.

With CXdb, you can:

- Debug an executable file
- Debug a core file
- Obtain a stack backtrace
- Attach to and debug a running process
- Debug at the source code or assembly code level
- Debug MPI applications with multiple processes using a single point of control

CXdb has an X/Motif graphical user interface and a command-line interface for supporting line-oriented terminals.

Using CXdb

To use CXdb in X window mode, follow these steps:

- Step 1.** Compile and link (using the compiler driver) your program with the `-g` option:

```
% f77 -g prog.f
```

In this example, the Exemplar linker is automatically used by `f77`. If you do not use the Exemplar linker, you will not be able to use CXdb on the resulting executable.

- Step 2.** Set your `DISPLAY` environment variable (if it is not already set). For example if your display's name is `mydisplay`, in C shell, enter:

```
% setenv DISPLAY mydisplay:0.0
```

- Step 3.** Invoke CXdb on the executable file:

```
% cxdb a.out
```

For additional information on using CXdb, refer to the `cxdb(1)` man page, the CXdb online help system, or the *CXdb Quick Reference*.

The CXperf profiler

CXperf is a performance analysis tool for monitoring program performance at user-selectable source code regions, such as routines and loops.

Types of performance data you can collect include:

- Wall clock time
- CPU time
- Dynamic call graph
- Execution counts
- Cache miss counts and latency time for memory accesses (available on V-Class only)

Features of CXperf include the ability to:

- Analyze profiling data in 2D and 3D graphs or text reports
- Clickback to source code during analysis
- View performance data for individual threads or summed across all threads of a process
- Profile MPI applications

For more information about these events, see the `cxperf(1)` man page or the CXperf online help system.

Using CXperf

The basic steps in the profiling process are as follows:

1. Prepare the program for profiling by compiling with the `+pa` option, by compiling at `+O2` or `+O3` with the `+pal` option, or by instrumenting it with the `cxoi` utility (refer to the `cxoi(1)` man page for more information).
2. Link the application with the `+pa` option or the `+pal` option using the Exemplar linker by means of the compiler driver.
3. Set your `DISPLAY` environment variable (if it is not already set). For example if your display's name is `mydisplay`, in C shell, enter:

```
% setenv DISPLAY mydisplay:0.0
```
4. Invoke CXperf (`cxperf`), specifying the name of the executable you want to profile:

```
% cxperf a.out
```
5. Select the metrics you want to collect and the source code regions (that is, routines and loops) at which you want them to be collected.
6. Run the program and generate a performance data file (PDF) by running the executable under the control of CXperf
or
writing instrumentation selections to the executable file, exiting CXperf, and then running the executable outside CXperf to generate performance data files for later analysis with CXperf.
7. Analyze the results in graphs and reports.

NOTE

The `+pa` and `+pal` options are not compatible with `-p` or `-G` options or with the `+O4` or `+Oall` optimization levels.

Debugging and profiling
Using CXperf

A

Environment variables

The Exemplar compilers honor many of the environment variables accepted by the standard Hewlett-Packard C and Fortran 77 compilers. This chapter describes some of those environment variables. See the following documents for more information on environment variables:

- `cc(1)` man page
- `f77(1)` man page
- *HP C/HP-UX Programmer's Guide*
- *HP C/HP-UX Reference Manual*
- *HP FORTRAN/9000 Programmer's Guide*
- *HP FORTRAN/9000 Programmer's Reference*

The following environment variables are discussed in this appendix:

- `CCOPTS`
- `FCOPTS`
- `MP_NUMBER_OF_THREADS`
- `TMPDIR`

CCOPTS

You can specify C compiler options by using the `CCOPTS` environment variable or by including them on the command line. The `CCOPTS` environment variable provides a convenient way for establishing default options for the C compiler's command line.

The syntax for setting the `CCOPTS` environment variable (in C shell notation) is:

```
% setenv CCOPTS [options] [ | [options]]
```

where

options Is a space-separated string of one or more compiler options

If you specify more than one option or you use the pipe (`|`), enclose the entire string in quotes.

The compiler places the options that appear before the pipe in front of the command-line options to the compiler. It then places the second group of options after any command-line options to the compiler.

Options that appear after the pipe in the `CCOPTS` variable override and take precedence over options supplied on the command line.

If the pipe is omitted, the compiler gets the value of `CCOPTS` and places its contents before any options on the command line.

For example, the following (in C shell notation)

```
% setenv CCOPTS -v
```

```
% cc -g prog.c
```

is equivalent to

```
% cc -v -g prog.c
```

Using the pipe, the following (in C shell notation)

```
% setenv CCOPTS "-v | +O1"
```

```
% cc +O2 prog.c
```

is equivalent to

```
% cc -v +O2 prog.c +O1
```

In the above example, level 1 optimization is performed because the +O1 option appearing after the pipe in CCOPTS takes precedence over the cc command-line options.

FCOPTS

You can specify Fortran 77 compiler options by using the `FCOPTS` environment variable or by including them on the command line. The `FCOPTS` environment variable provides a convenient way for establishing default options for the Fortran compiler's command line.

The syntax for setting the `FCOPTS` environment variable (in C shell notation) is:

```
% setenv FCOPTS [options] [ | [options]]
```

where

options Is a space-separated string of one or more compiler options

If you specify more than one option or you use the pipe (`|`), enclose the entire string in quotes.

The compiler places the options that appear before the pipe in front of the command-line options to the compiler. It then places the second group of options after any command-line options to the compiler.

Options that appear after the pipe in the `FCOPTS` variable override and take precedence over options supplied on the command line.

If the pipe is omitted, the compiler gets the value of `FCOPTS` and places its contents before any options on the command line.

Environment variables

FCOPTS

For example, the following (in C shell notation)

```
% setenv FCOPTS -v
```

```
% f77 -L prog.f
```

is equivalent to

```
% f77 -v -L prog.f
```

Using the pipe, the following (in C shell notation)

```
% setenv FCOPTS "-O | -lmylib"
```

```
% f77 -v prog.f
```

is equivalent to

```
% f77 -O -v prog.f -lmylib
```

MP_NUMBER_OF_THREADS

This environment variable specifies the number of processors to be used in executing programs compiled using `+Oparallel`. If not set, it defaults to the number of processors on the executing system. This variable is used at runtime by parallel programs compiled using either the C or the Fortran 77 compiler.

The following command line shows the C shell syntax to use when setting the variable to 8 processors:

```
% setenv MP_NUMBER_OF_THREADS 8
```

TMPDIR

The environment variable `TMPDIR` allows you to change the location of temporary files that the compiler creates and uses. Both the C and Fortran 77 compilers use this variable. The directory specified in `TMPDIR` replaces `/var/tmp` as the default directory for temporary files. The syntax for setting `TMPDIR` (in C shell notation) is:

```
% setenv TMPDIR altdir
```

where

<i>altdir</i>	Is the name of the alternative directory for temporary files
---------------	--

Index

A

ALIGN_CTI directive and pragma, 23
alignment
 and ALIGN_CTI directive and pragma, 23
assembler (as) examples, 50
automatic parallelization, 44

B

BEGIN_TASKS directive and pragma, 24
BLOCK_LOOP directive and pragma, 25
BLOCK_SHARED directive, 25
block_shared memory class, 25

C

c89 compiler, 9
cc compiler, 9
cc examples, 9
CCOPTS environment variable, 60
COMMON blocks
 Cray TASK COMMON, 38
compiler directives
 see directives
compiler pragmas
 see pragmas
compilers
 c89, 9
 cc, 9
 f77, 10
 fort77, 10
constants, INTEGER*8, 37
CPSlib, 47
 linking in, 47
Cray compatibility
 TASK COMMON, 38
CRITICAL_SECTION directive and pragma, 25
CXdb debugger, 14, 55
CXdb example, 55
CXperf overview, 57
CXperf profiler, 20, 56

D

+DA option, 21
dde debugger, 53
debugging, 2, 14, 54
 and +O0, 14, 54
 and the Exemplar linker, 54
 linker support, 51
 with CXdb, 54
 with dde, 53
 with xdb, 53
debugging option (-g), 14, 54
directives
 ALIGN_CTI, 23
 BARRIER, 23
 BEGIN_TASKS, 24
 BLOCK_LOOP, 25
 BLOCK_SHARED, 25
 CRITICAL_SECTION, 25
 DYNSEL, 25
 END_CRITICAL_SECTION, 26
 END_ORDERED_SECTION, 26
 END_PARALLEL, 26
 END_TASKS, 26
 FAR_SHARED, 26
 FAR_SHARED_POINTER, 27
 form, Exemplar compiler, 22
 Fortran compiler, 22
 GATE, 27
 LOOP_PARALLEL, 27, 35
 LOOP_PRIVATE, 29
 NEAR_SHARED, 29
 NEAR_SHARED_POINTER, 29
 NEXT_TASK, 30
 NO_BLOCK_LOOP, 30
 NO_DISTRIBUTE, 30
 NO_DYNSEL, 30
 NO_LOOP_DEPENDENCE, 30
 NO_LOOP_TRANSFORM, 31
 NO_PARALLEL, 31
 NO_SIDE_EFFECTS, 31
 NO_UNROLL_AND_JAM, 31
 NODE_PRIVATE, 31
 NODE_PRIVATE_POINTER, 32
 ORDERED_SECTION, 32
 PARALLEL, 33
 PARALLEL_PRIVATE, 33
 PREFER_PARALLEL, 34

SAVE_LAST, 35
SCALAR, 35
SYNC_ROUTINE, 36
TASK_PRIVATE, 36
THREAD_PRIVATE, 36
THREAD_PRIVATE_POINTER, 36
UNROLL_AND_JAM, 37
+DS option, 21
dynamic selection
 and DYNSEL directive and pragma, 25
 and +O[no]dynsel compiler option, 16
DYNSEL directive and pragma, 25

E

ELF (Executable and Linking Format), 51
END_CRITICAL_SECTION directive and pragma, 26
END_ORDERED_SECTION directive and pragma, 26
END_PARALLEL directive and pragma, 26
END_TASKS directive and pragma, 26
environment variables
 CCOPTS, 60
 FCOPTS, 62
 MP_NUMBER_OF_THREADS, 18, 59
 TMPDIR, 64
equivalences, and -Wc,-local_equivs, 40
examples
 assembler, 50
 cc, 9
 CXdb, 55
 f77, 10
 linker, 52
Exemplar programming model, 1, 16, 22
extensions
 INTEGER*8, 37
 INTEGER*8 constants, 37
 LOGICAL*8, 37
 TASK COMMON, 38

F

f77 compiler, 10
f77 examples, 10
FAR_SHARED directive, 26
FAR_SHARED_POINTER directive, 27
FCOPTS environment variable, 62
fort77 compiler, 10
Fortran intrinsics, 39
Fortran language extensions, 37

G

-g option, 14, 54

H

header file
 pthread.h, 46
 spp_prog_model.h, 22

I

-I8 option, 14
INTEGER*8 constants, 37
INTEGER*8 extension, 37
intrinsics, 39

L

linker
 examples, 52
LOGICAL*8 extension, 37
loop blocking
 and +O[no]loop_block option, 17
Loop Report, 19
loop unroll and jam
 and +O[no]loop_unroll_jam option, 17
LOOP_PARALLEL directive and pragma, 27
LOOP_PARALLEL pragma, 35
LOOP_PRIVATE directive and pragma, 29

M

memory classes
 block_shared, 25
 FAR_SHARED_POINTER directive, 27
 near_shared_pointer, 29
 node_private_pointer, 32
 thread_private_pointer, 36
MP_NUMBER_OF_THREADS environment variable,
 18, 59, 64
MPI message passing, 45

N

NEAR_SHARED directive, 29
NEAR_SHARED_POINTER directive, 29
NEXT_TASK directive and pragma, 30
NO_BLOCK_LOOP directive and pragma, 30
NO_DISTRIBUTE directive and pragma, 30
NO_DYNSEL directive and pragma, 30
NO_LOOP_DEPENDENCE directive and pragma, 30
NO_LOOP_TRANSFORM directive and pragma, 31, 35
NO_PARALLEL directive and pragma, 31
NO_SIDE_EFFECTS directive and pragma, 31
NO_UNROLL_AND_JAM directive and pragma, 31
NODE_PRIVATE directive, 31
NODE_PRIVATE_POINTER directive, 32

O

- O option, 3
- +O0 option, 2
- +O1 option, 3
- +O2 option, 3
- +O3 option, 5
- +O4 option, 5
- +O[no]aggressive option, 6
- +O[no]all option, 6
- +O[no]autopar option, 15
- +O[no]dataprefetch option, 15
- +O[no]exemplar_model option, 12, 16
- +O[no]fail_safe option, 6
- +O[no]info option, 7, 19
- +O[no]limit option, 7
- +O[no]loop_block option, 17
- +O[no]loop_transform option, 8
- +O[no]loop_unroll option, 8
- +O[no]loop_unroll_jam option, 17
- +O[no]parallel option, 11, 18, 44
- optimization reports
 - Loop Report, 19
 - overview, 19
- options
 - +DA, 21
 - +DS, 21
 - g, 14, 54
 - I8, 14
 - O, 3
 - +O0, 2
 - +O1, 3
 - +O2, 3
 - +O3, 5
 - +O4, 5
 - +O[no]aggressive, 6
 - +O[no]all, 6
 - +O[no]autopar, 15
 - +O[no]dataprefetch, 15
 - +O[no]exemplar_model, 12, 16
 - +O[no]fail_safe, 6
 - +O[no]info, 7, 19
 - +O[no]limit, 7
 - +O[no]loop_block, 17
 - +O[no]loop_transform, 8
 - +O[no]loop_unroll, 8
 - +O[no]loop_unroll_jam, 17
 - +O[no]parallel, 11, 18, 44
 - +O[no]report, 19
 - +Oreport=all, 19
 - +Oreport=loop, 19
 - +Oreport=private, 19
 - +O[no]sharedgra, 19
 - +O[no]size, 8
 - +pa, 20, 57
 - +pal, 20, 57

- +tm target, 20
- ORDERED_SECTION directive and pragma, 32
- +O[no]report option, 19
- +Oreport=all option, 19
- +Oreport=loop option, 19
- +Oreport=private option, 19
- +O[no]sharedgra option, 19
- +O[no]size option, 8

P

- +pa option, 20, 57
- +pal option, 20, 57
- PARALLEL directive and pragma, 33
- parallel regions
 - and END_PARALLEL directive and pragma, 26
 - and PARALLEL directive and pragma, 33
 - privatizing data in, 33
- PARALLEL_PRIVATE directive and pragma, 33
- parallelization, 43
 - begin_tasks directive/pragma, 24
 - CPSlib, 47
 - HP MPI, 45
 - loop_parallel directive/pragma, 27
 - +O3, 5, 44
 - +O[no]autopar, 15
 - +O[no]parallel, 11, 18, 44
 - optimization level +O3, 5, 44
 - parallel directive/pragma, 33
 - prefer_parallel directive/pragma, 34
 - Pthreads, 46
- pragmas
 - align_cti, 23
 - begin_tasks, 24
 - block_loop, 25
 - C compiler, 22
 - critical_section, 25
 - dynsel, 25
 - end_critical_section, 26
 - end_ordered_section, 26
 - end_parallel, 26
 - end_tasks, 26
 - form, Exemplar compiler, 22
 - gate, 27
 - loop_parallel, 27, 35
 - loop_private, 29
 - next_task, 30
 - no_block_loop, 30
 - no_distribute, 30
 - no_dynsel, 30
 - no_loop_dependence, 30
 - no_loop_transform, 31
 - no_parallel, 31
 - no_side_effects, 31
 - no_unroll_and_jam, 31
 - ordered_section, 32

- parallel, 33
- parallel_private, 33
- prefer_parallel, 34
- reduction, 35
- save_last, 35
- scalar, 35
- sync_routine, 36
- task_private, 36
- unroll_and_jam, 37
- Predefined symbols, 41
 - __HP_CXD_SPP=1, 41
 - __REENTRANT=1, 41
- PREFER_PARALLEL directive and pragma, 34
- Privatization Table, 19
- profiler
 - CXperf, 56
- profiling, 20
 - and the Exemplar linker, 57
 - linker support, 51
- programming model, 1, 16, 22
- pthread.h header file, 46
- Pthreads, 46
 - linking in, 46

R

- REDUCTION pragma, 35
- region parallelization
 - and PARALLEL directive and pragma, 33
- register allocation, 19

S

- SAVE_LAST directive and pragma, 35
- SCALAR directive and pragma, 35
- SOM (Standard Object Module), 51
- spp_prog_model.h header file, 22
- statements
 - TASK COMMON, 38
- SYNC_ROUTINE directive and pragma, 36

T

- TASK COMMON extension, 38
- TASK COMMON statement, 38
 - form, 38
- task private data, 36
- TASK_PRIVATE directive and pragma, 36
- THREAD_PRIVATE directive, 36
- THREAD_PRIVATE_POINTER directive, 36
- +tm target, 20
 - K7200 target value, 20
 - K8000 target value, 20
 - S2000 target value, 20
 - spp1200 target value, 20
 - spp1600 target value, 20
 - V2200 target value, 20
 - X2000 target value, 20
 - and +DA, 21
 - and +DS, 21
- TMPDIR environment variable, 64

U

- UNROLL_AND_JAM directive and pragma, 37

X

- xdb debugger, 53