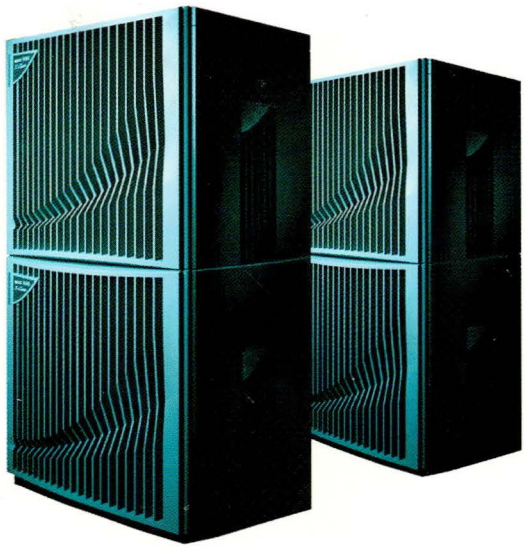
HEWLETT® PACKARD

S-Class and
X-Class Servers



Exemplar C and Fortran 77 Programmer's Guide

First Edition

Hewlett-Packard Company
Convex Division
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America

Exemplar C and Fortran 77 Programmer's Guide

S-Class and X-Class Servers

B5600-90002

First Edition

January 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

Exemplar C and Fortran 77 Programmer's Guide

S-Class and X-Class Servers

B5600-90002

Notice

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.

This entire book is recyclable.

Printed in the United States of America

Revision Information for

Exemplar C and Fortran 77 Programmer's Guide

S-Class and X-Class Servers

Edition	Document No.	Description
First	B5600-90002	Initial release January 1997

Contents

Figures

Tables

How to use this guide

Purpose and audience

This guide describes how to use the Exemplar C and Fortran 77 compilers. It describes the differences between the Exemplar compilers running on SPP-UX and the standard Hewlett-Packard (HP) compilers on which they are based. All Exemplar-specific features are described in detail. This guide also describes the differences between the Exemplar compilers and the SPP1000-Series compilers (/usr/convex/bin/cc and /usr/convex/bin/fc).

The audience for this book is the experienced C or Fortran 77 programmer who has a basic familiarity with SPP-UX, HP-UX, or Unix and is using V1.1 (or higher) of one of the Exemplar compilers to produce applications for SPP-UX V5.1 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. (S2000 servers are also known as S-Class servers; X2000 servers are also known as X-Class servers.)

Organization

This guide is organized as follows:

- Chapter 1 introduces the Exemplar compilers and provides some standard HP compiler information and examples.
- Chapter 2 describes the Exemplar extensions to the standard HP compilers, including options, directives and pragmas, Fortran 77 language extensions, and predefined preprocessor symbols.
- Chapter 3 describes differences between the Exemplar compilers and the SPP1000-Series compilers in terms of available options, directives and pragmas, and Fortran 77 language extensions.
- Chapter 4 provides a quick overview of how the Exemplar assembler and linker utilities differ from the HP-UX versions on which they are based. This chapter also describes SOM and ESOM files and the linker support for the CXdb debugger and the CXpa profiler.
- Chapter 5 gives an overview of the CXdb debugger and the CXpa profiler.
- Chapter 6 explores system utilities that can be helpful in programming an SPP-UX system.
- Appendix A describes common environment variables that the Exemplar compilers or their applications honor.

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 ...*]

Exemplar compilers

The Hewlett-Packard Exemplar C and Fortran 77 compilers are based on 10.2*x* releases of the corresponding Hewlett-Packard compilers. The Exemplar compilers support the creation, debugging, and profiling of thread-parallel applications running on SPP-UX V5.1 or higher.

SPP1000-Series compilers

The phrase *SPP1000-Series compilers* refers to the /usr/convex/bin/cc and /usr/convex/bin/fc compilers. Hewlett-Packard now ships the Exemplar compilers in place of the SPP1000-Series compilers.

Standard HP compilers

The phrase *standard HP compilers* refers to the C and Fortran 77 compilers developed by Hewlett-Packard Company.

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.
- *CXpa Reference* (B5639-90002)—This book describes the CXpa performance analyzer.
- *Exemplar Programming Guide* (B5600-90001)—This book describes efficient shared-memory programming techniques for the SPP1200 systems, SPP1600 systems, and the S2000 and X2000 technical servers.
- *HP MPI User's Guide* (B6011-90001)—This book discusses message-passing programming using Hewlett-Packard's Message-Passing Interface library.
- *HP PVM User's Guide* (B5885-90001)—This book discusses message-passing programming using Hewlett-Packard's Parallel Virtual Machine 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 Notes* (B5876-90002)—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.

- *SPP-UX System Administrator's Guide* (B5655-90002)—This manual describes fundamental concepts and tasks associated with setting up and maintaining SPP1200 systems, SPP1600 systems, S2000 servers, and X2000 servers.
- The following man pages:
  - as(1)
  - cc(1)
  - chatr(1)
  - cnx_ps(1)
  - cxdb(1)
  - cxoi(1)
  - cxpa(1)
  - f77(1)
  - file(1)
  - largefiles(1m)
  - ld(1)
  - make(1)
  - mpa(1)
  - nm(1)
  - pot(1)
  - scm(1)
  - size(1)
  - sod(1)
  - sysinfo(1)
  - syspic(1)
  - top(1)

Ordering documents

To order additional copies of this document or other documents listed in the "Associated documents" section, send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (xxxxx-9xxxx number) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

Within the continental U.S., call 1-800-952-0379.

From Canada, call 1-800-345-2384.

All other locations, contact the local Hewlett-Packard office.

You can also use the `contact` utility, if you would like to report any problems you may have with the Exemplar compilers or the documentation. For more information refer to the contact(1) man page.

Introduction 1

This chapter introduces the Hewlett-Packard Exemplar C and Fortran 77 compilers. These compilers are based on the 10.2*x* releases of the standard Hewlett-Packard C and Fortran 77 compilers and are designed for creating applications for SPP-UX V5.1 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* and the *HP PVM 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 xvi for a list of additional documentation.

Note

The exact optimizations performed by the SPP1000-Series compilers (/usr/convex/bin/fc, /usr/convex/bin/cc) are not available in the Exemplar compilers. The Exemplar compilers perform the optimizations supported by the standard HP compilers. In many cases, these optimizations are similar. Also, 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	Re-arranges the order in which instructions execute in a loop to prevent processor stalls

Table 3 Optimizations performed at +O2 —**(continued)**

Optimization	Description
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), directive-specified blocking, 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)
- +Oregionsched
- +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 21.

+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*" on page 7 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, 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]parallel_env

Note

Do not use the +Oparallel_env option unless you are creating a process-based parallel application. Applications created using the Exemplar programming model are thread-based parallel.

This option compiles for a parallel [serial] execution environment. The +Oparallel_env option does not request parallelization for the target source; rather, it ensures a consistent execution environment for all files in a parallel program. This option is only supported for applications using process-based parallelism. If you want to compile an application for process-based parallel execution, you must compile all of its files with +Oprocess_threads and either +Oparallel or +Oparallel_env. Do not use +Oparallel_env with +Oparallel.

+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

There are two C compilers, as 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 of the C compiler options

files

is a space-delimited 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.

This command

```
% cc -g +O0 prog.c
```

shows the debugging option (`-g`) and the request of level 0 optimizations (`+O0`).

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

Using the Fortran 77 compiler

There are two Fortran 77 compilers, as 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 of the Fortran 77 compiler options

files

is a space-delimited 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 20 for more information.
`+Odataprefetch`	Prefetch data referenced in loops. See the section "`+O[no]dataprefetch`" on page 17 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. (Available only on S2000 and X2000 servers.)
`+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 17 for more information.
`+Onolimit`	Do not suppress optimizations that significantly increase compile-time or consume large amounts of memory. See the section "`+O[no]limit`" on page 8 for more information.
`+Onodepar`	Enable directive-specified, node-level parallelism. See the section "`+O[no]nodepar`" on page 19 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) man page for more information. (Available only in C.)

Table 4 Options to get you started —**(continued)**

Option	Description
`+Oreport`	Display optimization reports. See the section "`+O[no]report[=`*report_type*`]`" on page 21 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) Underflows are exceptions, by default; this option avoids exceptions so that underflows just generate zeros.

*Assuming `+Onoexemplar_model` is not also specified (See the section "`+O[no]exemplar_model`" on page 18 for more information.)

Exemplar extensions 2

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.

The `-g` option is ignored at optimization levels greater than `+O0`. Also, `-g` is ignored with `-O` because it implies `+O2`.

`-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 places 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. This option is effective only on S2000 and X2000 servers.

+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 23 for additional information.

The +Oexemplar_model option implies the +Okernel_threads option.

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. +Onoexemplar_model can be used with either +Okernel_threads (the default) or +Oprocess_threads.

+Okernel_threads

The +Okernel_threads option causes the compiler to use a thread-based model of parallelism. The Exemplar programming model requires thread-based parallelism. This option is available at all optimization levels and is enabled by default.

Alternatively, you can specify process-based parallelism by using the +Oprocess_threads option. See the section "+Oprocess_threads" on page 21 for more information.

+Okernel_threads can be used with either +Oexemplar_model or +Onoexemplar_model.

+O[no]nodepar

The +Ononodepar option disables node-parallelism by causing the compiler to generate code for a single-node machine. When this option is used, serial code is generated for node-parallel constructs. Specifying the +Ononodepar option prevents the compiler from implementing node-parallelism, but allows the implementation of both automatic and directive-specified thread-parallelism.

The +Onodepar option causes the compiler to perform node-parallelism where it has been specified using the nodes attribute with the loop_parallel, prefer_parallel, parallel, or begin_tasks directives or pragmas. Also, the +Onodepar option causes the compiler to honor the node_trip_count attribute to the dynsel directive or pragma.

The +O[no]nodepar option is effective only when specified with the +Oparallel option at +O3 and above. The default is +Ononodepar.

+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.

There are three ways to specify the number of processors used in executing your parallel programs:

- The first method uses the environment variable MP_NUMBER_OF_THREADS, which is used at runtime. 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 subcomplex where the program is executing. If MP_NUMBER_OF_THREADS is not set, your program runs on the number of processors in the subcomplex where it is executing. (See the section "Subcomplexes" on page 71 for information on subcomplexes.)
- The second method for selecting the number of processors to use when running parallel code involves using the mpa utility, which provides more control (than MP_NUMBER_OF_THREADS) over the attributes in a parallel program. See the section "Using the mpa utility" on page 75 or the mpa(1) man page for more information.
- The third method involves using the +min and +max linker options. See the ld(1) man page for more information.

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.

+Oprocess_threads

The +Oprocess_threads option causes the compiler to use process-based parallelism. Process-based parallelism is used by the standard HP compilers.

+Oprocess_threads implies +Onoexemplar_model, which causes directives and pragmas from the Exemplar programming model to be ignored.

If you specify both +Oexemplar_model and +Oprocess_threads, +Oprocess_threads is ignored with a warning, and +Okernel_threads is selected.

+Okernel_threads is the default. See the section "+Okernel_threads" on page 19 for more information.

+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. See the *Exemplar Programming Guide* for more information on the optimization reports.

The option +Oinfo displays additional 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 compiler add instrumentation (additional information) to an executable file for the CXpa performance analyzer to read. 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 CXpa.

+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

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. If you are manually linking your application, you have to specify the +tm *target* option.

Using the +tm *target* option implies +DA and +DS settings as described in Table 5. +DA*architecture* causes the compiler to generate code for the architecture specified by *architecture*. +DS*model* 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

target value specified	+DA*architecture* implied	+DS*model* implied
spp1200	1.1	1.1
spp1600	1.1	1.1
S2000	2.0	2.0
X2000	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-specification*

The form of an Exemplar C pragma is:

#pragma _CNX *directive-specification*

where

directive-specification

is one 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-delimited list of names. These names can be variables, arrays, or COMMON blocks. In the case of a COMMON block, its name must be enclosed within slashes. The occurrence of a 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, or SMP, running a single image of the operating system microkernel.)

The CTIcache is 64 bytes on SP1200 and SPP1600 systems. On X2000 servers, the CTIcache is 32 bytes. (S2000 servers do not use a CTIcache.) 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 subcomplex. If the user-specified size is not an integral multiple of page size × `num_nodes()`, then the compiler automatically rounds it 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.

`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 subcomplex. 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`
- `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*

`ivar =` *indvar* 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 subcomplex 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, interchange, 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.

`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`
- `ordered`
- `max_threads=`*m*
- `chunk_size=`*n*
- `threads, ordered`
- `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*

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.

`save_last[(list)]`

This directive or pragma specifies that the variables in the comma-delimited *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 its value that is generated 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, interchange, 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.

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 date type allocates storage for 8-byte logical data.

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

cbn

is a symbolic name for a TASK COMMON block. Unnamed TASK COMMON blocks are not allowed.

nlist

is a list of variable names, array names, and array declarators. These variables cannot appear in a DATA statement, but otherwise can be used like any variables in COMMON 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`
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 if `+Oparallel` is specified with either `+O3` or `+O4`.

Large files support

The SPP-UX operating system and the Exemplar compilers support *large files*. A large file is a file that is greater than $2^{31} - 1$ bytes in size (approximately 2 gigabytes).

Several SPP-UX utilities have been modified to function properly on large files. See the largefiles(1m) man page for information on the modified utilities and on compiler support for large files.

Thread-based parallelism

This section discusses the various methods you can use to create a parallel executable. A parallel executable does not necessarily execute in parallel; however, when a parallel executable is run, the proper parallel environment is always set up—regardless of whether the executable runs serially or in parallel.

There are three ways to create a parallel executable (an executable in which the parallel flag is set):

- Using the `+Oparallel` compiler option at `+O3` and above
- Using linker options (`+min`, `+max`, `+tnode`, `+over`, or `+parallel`)
- Using `mpa` (`-min`, `-max`, `-over`, or `-parallel`)

Applications that rely strictly on the message-passing model to achieve parallelism do not need the parallel flag set. Message-passing applications that use multilevel parallelism do, however, need the parallel flag set. For information on developing parallel applications that use message-passing, see the *HP MPI Users' Guide* or the *HP PVM User's Guide*.

See the section "Using the `file` utility" on page 74 for information on determining if a file is a parallel executable.

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 subcomplex 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 20 or to the *Exemplar Programming Guide*.

Using linker options

Specifying the `+parallel` linker option sets the parallel flag in the ESOM auxiliary header. (See the section "SOM vs. ESOM" on page 62 for information on SOM and ESOM files.) When linking using the compiler driver, if the `+Oparallel` compiler option is used (at `+O3` or above), the `+parallel` option is automatically passed to the linker.

If any of the object files being linked are already parallel, you do not need to specify `+parallel`. Also, if you already specified `+min` *n*, `+max` *n* (*n* is the number of processors to use), `+tnode` *m* (*m* is the maximum number of threads to allocate per hypernode) or `+over`, you do not need to specify `+parallel`. See the ld(1) man page for more information.

Using the `mpa` utility

The `mpa` (modify program attributes) utility allows you to set the parallel flag in the ESOM auxiliary header using the `-parallel` option. It also allows you to set the number of processors used when executing a parallel program (using `-min` *n* or `-max` *n*, where *n* is the number of processors to use). For information on other features, refer to the section "Using the `mpa` utility" on page 75 or to the mpa(1) man page.

Migrating to the Exemplar compilers

3

This chapter provides transition information for users moving from the SPP1000-Series compilers (/usr/convex/bin/cc, /usr/convex/bin/fc) to the Exemplar compilers, which are based on the standard HP compilers. Options, directives, and pragmas available in the SPP1000-Series compilers are mapped to the equivalent features in the Exemplar compilers. In addition, this chapter lists the language extensions from the SPP1000-Series compilers that the Exemplar compilers support. Finally, this chapter covers changes in accessing CPSlib.

Compiler options

This section lists the compiler options from the SPP1000-Series compilers that are supported by the Exemplar compilers or are unimplemented.

In Table 7, the compiler options in the left column were available in the SPP1000-Series compilers (/usr/convex/bin/cc, /usr/convex/bin/fc). The middle column lists corresponding or similar options in the Exemplar compilers or states that the option is unimplemented. The right column points to additional information on the Exemplar compiler options.

Note

Functionality may differ between corresponding options.

The following table does not list options (such as `-o` and `-c`) that are common across a number of compilers.

Table 7 Mapping of compiler options

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
`-72`	Default	
`-a1`	unimplemented option	
`-alias addr`	unimplemented option	
`-alias array_args`	`+Onoparmsoverlap`	cc(1) man page
`-alias cautious`	unimplemented option	
`-alias global`	`+Optrs_to_globals`	cc(1) man page
`-alias no_addr`	unimplemented option	
`-alias no_global`	`+Onoptrs_to_globals`	cc(1) man page
`-alias ptr_args`	unimplemented option	
`-alias restrict_args`	unimplemented option	
`-alias standard`	`+Optrs_strongly_typed`	cc(1) man page
`-alias worst`	`+Onoptrs_ansi`	cc(1) man page
`-align cache`	unimplemented option	
`-align cache_check`	unimplemented option	
`-align cseries`	unimplemented option	
`-align cti`	unimplemented option	
`-align spp`	`+A8` (Fortran only)	f77(1) man page

Table 7 Mapping of compiler options —(continued)

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
`-ansi77`	`-A`	f77(1) man page
`-ansi90`	unimplemented option	
`-blockloop` *n*	unimplemented option	
`-br`	unimplemented option	
`-cache` *n*	unimplemented option	
`-cfc`	Cray Fortran extensions are enabled by default	*HP FORTRAN/9000 Programmer's Reference*
`-com`	similar to `-v`	cc(1) man page, f77(1) man page
`-cs`	`-C`	f77(1) man page
`-ctifiles`	unimplemented option	
`-cxdb`	`-g`	Chapter 2, "Exemplar extensions"
`-cxpa`	`+pa`	Chapter 2, "Exemplar extensions"
`-cxpab`	unimplemented option	
`-cxpalib`	unimplemented option	
`-cxpamon`	unimplemented option	
`-cxpar`	`+pa`	Chapter 2, "Exemplar extensions"
`-d` *name* `[={w|e}]`	unimplemented option	
`-dc`	`-D`	f77(1) man page
`-ds`	`+Odynsel`	Chapter 2, "Exemplar extensions"
`-errnames`	unimplemented option	
`-ext`	similar to `-Ae`	cc(1) man page
`-extern distinct`	unimplemented option	
`-extern same`	Default	
`-F66`	`-w66`	f77(1) man page
`-fd`	see `+f` and `+r`	cc(1) man page

Table 7 Mapping of compiler options —(continued)

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
-fl	+Oloop_transform	Chapter 1, "Introduction"
-float dp_const	unimplemented option	
-float sp_const	see +f and +r	cc(1) man page
-float dp_ops	unimplemented option	
-float sp_ops	see +f and +r	cc(1) man page
-gs	+Osharedgra	Chapter 2, "Exemplar extensions"
-i1	unimplemented option	
-i2	-I2	f77(1) man page
-i4	-I4	f77(1) man page
-i8	-I8	Chapter 2, "Exemplar extensions"
-il	unimplemented option	
-ipo	+O4	Chapter 1, "Introduction"
-is *directory*	unimplemented option	
-LST	-L	f77(1) man page
-LSTI	unimplemented option	
-mo	similar to +Onofltacc	cc(1) man page, f77(1) man page
-mrl	similar to +Onolimit +Onosize	cc(1) man page, f77(1) man page
-na	unimplemented option	
-nbr	unimplemented option	
-nds	+Onodynsel	Chapter 2, "Exemplar extensions"
-nfl	+Onoloop_transform	cc(1) man page, f77(1) man page
-nga	unimplemented option	
-ngr	unimplemented option	

Table 7 Mapping of compiler options —**(continued)**

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
-ngs	+Onosharedgra	Chapter 2, "Exemplar extensions"
-nmo	similar to +Ofltacc	cc(1) man page, f77(1) man page
-no	+O0 (default)	Chapter 1, "Introduction"
-noautopar	+Onoautopar	Chapter 2, "Exemplar extensions"
-noblock	unimplemented option	
-nof90	unimplemented option	
-nonodepar	Default (Use +Onodepar to enable directive-specified, node-parallelism.)	Chapter 2, "Exemplar extensions"
-nopeel	unimplemented option	
-noptst	unimplemented option	
-nore	unimplemented option	
-nosc	unimplemented option	
-noU77	Default (Use +U77 to get libU77 routines.)	f77(1) man page
-nptr	unimplemented option	
-nsr	unimplemented option	
-nuj	unimplemented option	
-nur	+Onoloop_unroll	Chapter 1, "Introduction"
-nv	Default is no report	
-nw	-w	cc(1) man page, f77(1) man page
-O (same as -O2)	-O (same as +O2)	Chapter 1, "Introduction"
-O0	unimplemented option	
-O1	similar to +O2	Chapter 1, "Introduction"

Table 7 Mapping of compiler options —**(continued)**

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
`-O2`	similar to `+O3`	Chapter 1, "Introduction"
`-O3`	similar to `+O3 +Oparallel`	Chapter 2, "Exemplar extensions"
`-or` *table*	`+Oreport[=`*report_type*`]`	Chapter 2, "Exemplar extensions"
`-p`	`-p` (Using `prof` for multithreaded applications is not supported)	
`-p8`	`$AUTODBL DBL` source code directive	*HP FORTRAN/9000 Programmer's Reference*
`-parens explicit`	unimplemented option	
`-parens ignore`	unimplemented option	
`-parens implicit`	unimplemented option	
`-pcc`	`-Ac`	cc(1) man page
`-pd8`	similar to `-I8` with `$AUTODBL DBL DBL4` source code directive	*HP FORTRAN/9000 Programmer's Reference*, and Chapter 2, "Exemplar extensions"
`-peel`	unimplemented option	
`-peelall`	unimplemented option	
`-pg`	`-G` (Using `gprof` for multithreaded applications is not supported)	
`-pl` *n*	`$LINES` *n* source code directive	*HP FORTRAN/9000 Programmer's Reference*
`-ppu`	`+ppu`	f77(1) man page
`-ptst`	unimplemented option	
`-ptstall`	unimplemented option	
`-pw` *n*	unimplemented option	
`-r4`	`-R4`	f77(1) man page
`-r8`	`-R8`	f77(1) man page
`-re`	unimplemented option	

Table 7 Mapping of compiler options —(continued)

SPP1000-Series compiler option*	Exemplar compiler option	For more information, see
`-sc`	unimplemented option	
`-sfc`	Sun Fortran extensions are enabled by default	*HP FORTRAN/9000 Programmer's Reference*
`-sr`	unimplemented option	
`-std`	`-Aa`	cc(1) man page
`-string read_only`	`+ESlit`	cc(1) man page
`-string write_only`	Default	
`-tm` *target*	`+tm` *target*	Chapter 2, "Exemplar extensions"
`-tri off`	unimplemented option	
`-tri on`	Default	
`-uj`	unimplemented option	
`-ujn` *n*	unimplemented option	
`-uo`	unimplemented option	
`-ur`	`+Oloop_unroll`	Chapter 1, "Introduction"
`-urn` *n*	`+Oloop_unroll=`*n*	Chapter 1, "Introduction"
`-vfc`	VAX Fortran extensions are enabled by default	*HP FORTRAN/9000 Programmer's Reference*
`-vn`	Use the `what` command on the executable (either /opt/ansic/bin/cc or /opt/fortran/bin/f77) to display version info for an Exemplar compiler	what(1) man page
`-xr`	`+R`	f77(1) man page
`-xra`	unimplemented option	

*SPP1000-Series compiler refers to /usr/convex/bin/cc and /usr/convex/bin/fc

Directives and pragmas

This section lists the directives and pragmas from the SPP1000-Series compilers (/usr/convex/bin/cc and /usr/convex/bin/fc) that are supported by the Exemplar compilers or are unimplemented.

The forms of the directives and pragmas in the SPP1000-Series compilers are accepted by the Exemplar compilers.

The form of an SPP1000-Series Fortran compiler directive is:

`C$DIR [CSERIES|SPP]` *directive-specification*

The form of an SPP1000-Series C pragma is:

`#pragma _CNX [CSERIES|SPP]` *directive-specification*

If `CSERIES` is specified, the directive is discarded by the compiler. See the section "Exemplar compiler directives and pragmas" on page 23 for information on the recommended forms for the Exemplar compilers.

Table 8 lists the compiler directives and pragmas and states whether they exist in the Exemplar compilers. Words in *italics* indicate user-supplied information. For the directives and pragmas that are implemented, descriptions are given in Chapter 2. However, for more information on using the directives and pragmas, see the *Exemplar Programming Guide.*

Table 8 Compiler directives/pragmas

Directive/pragma	Status
`barrier(`*namelist*`)` directive in Fortran `barrier_t` typedef in C	Implemented*
`begin_tasks[(`*attribute_list*`)]`	Implemented*
`block_loop[(block_factor=`*n*`)]`	Implemented*
`block_shared(`*allocatable_array_namelist*`)`	Implemented*
`critical_section[(`*gate_var*`)]`	Implemented*
`dynsel[(`*trip_count=n*`)]`	Implemented*
`end_ordered_section`	Implemented*
`end_tasks`	Implemented*
`far_shared(`*namelist*`)` directive in Fortran `far_shared` storage class specifier in C	Implemented*
`gate(`*namelist*`)` directive in Fortran `gate_t` typedef in C	Implemented*
`loop_parallel[(`*attribute_list*`)]`	Implemented*
`loop_private(`*namelist*`)`	Implemented*
`near_shared(`*namelist*`)` directive in Fortran `near_shared` storage class specifier in C	Implemented*
`next_task`	Implemented*
`no_block_loop`	Implemented*
`no_distribute`	Implemented*
`no_dynsel`	Implemented*
`no_fuse`	Unimplemented directive/pragma
`no_loop_dependence(`*namelist*`)`	Implemented*
`no_parallel`	Implemented*
`no_peel`	Unimplemented directive/pragma
`no_promote_test`	Unimplemented directive/pragma

Table 8 Compiler directives/pragmas —**(continued)**

Directive/pragma	Status
`no_side_effects(`*funclist*`)`	Implemented*
`no_unroll`	Unimplemented directive/pragma
`no_unroll_and_jam`	Unimplemented directive/pragma
`node_private(`*namelist*`)` directive in Fortran `node_private` storage class specifier in C	Implemented*
`opt_level (`*level*`)`	Similar to `opt_level` *level_arg* in C, `OPTIMIZE` *level_arg* in Fortran**
`ordered_section (`*gate_var*`)`	Implemented*
`peel`	Unimplemented directive/pragma
`peel_all`	Unimplemented directive/pragma
`prefer_fuse`	Unimplemented directive/pragma
`prefer_parallel [ (`*attribute_list*`) ]`	Implemented*
`promote_test`	Unimplemented directive/pragma
`promote_test_all`	Unimplemented directive/pragma
`row_wise`	Unimplemented directive/pragma
`save_last [ (`*list*`) ]`	Implemented*
`scalar`	Implemented*
`sync_routine (`*routinelist*`)`	Implemented*
`task_private (`*namelist*`)`	Implemented*
`thread_private (`*namelist*`)` directive in Fortran `thread_private` storage class specifier in C	Implemented*
`unroll [ (unroll_factor=`*n*`) ]`	Unimplemented directive/pragma
`unroll_and_jam [ (unroll_factor=`*n*`) ]`	Unimplemented directive/pragma

* For more information on implemented directives and pragmas, see Chapter 2, "Exemplar extensions" or the *Exemplar Programming Guide.*

** For more information on the `opt_level` pragma and the `OPTIMIZE` directive, see the *Exemplar Programming Guide.*

Fortran 77 language extensions

This section lists the Fortran 77 language extensions from the SPP1000-Series Fortran compiler (/usr/convex/bin/fc) that are supported by the Exemplar Fortran compiler or are unimplemented.

Table 9 lists the extensions that are supported in the Exemplar Fortran 77 compiler. Because some of these extensions are available in the standard HP Fortran compiler, the functionality and syntax may differ slightly from the SPP1000-Series Fortran compiler. Unless noted otherwise, see the *HP FORTRAN/9000 Programmer's Reference* for information on using these extensions.

Table 9 Fortran 77 language extensions

Extension	Status
$ edit descriptor	Implemented
* edit descriptor	Implemented
%REF	Implemented
%VAL	Implemented
.XOR.	Implemented
ACCEPT	Implemented

ALLOCATABLE
This statement is implemented but has a different syntax.
/usr/convex/bin/fc required parentheses:
`INTEGER*8 A(:)`
`ALLOCATABLE (A)`
f77 does not allow parentheses:
`INTEGER*8 A(:)`
`ALLOCATABLE A`
	Implemented

ALLOCATE	Implemented
AUTOMATIC	Implemented
BLOCKSIZE keyword	Implemented
BUFFERIN	Unimplemented extension
BUFFEROUT	Unimplemented extension
CARRIAGECONTROL keyword	Implemented
COMPLEX*8	Implemented
COMPLEX*16	Implemented
DEFAULTFILE keyword	Unimplemented extension

Table 9 Fortran 77 language extensions —(continued)

Extension	Status
DECODE	Implemented
DISPOSE keyword	Unimplemented extension
DO WHILE	Implemented
DOUBLE COMPLEX	Implemented
ENCODE	Implemented
END DO	Implemented
FIND	Unimplemented extension
IMPLICIT NONE	Implemented
INCLUDE	Implemented
INTEGER*1	Unimplemented extension
INTEGER*2	Implemented
INTEGER*4	Implemented
INTEGER*8 (See the section "INTEGER*8" on page 38)	Implemented
IOSTAT keyword	Implemented
LOC	Implemented
LOGICAL*1	Implemented
LOGICAL*2	Implemented
LOGICAL*4	Implemented
LOGICAL*8 (See the section "LOGICAL*8" on page 38)	Implemented
MAXREC keyword	Unimplemented extension
NAME keyword	Implemented
NAMELIST	Implemented
NOSPANBLOCKS keyword	Unimplemented extension
O edit descriptor	Unimplemented extension
PARAMETER (FORTRAN 66/VAX version)	Implemented
POINTER	Implemented
Q edit descriptor	Implemented
R edit descriptor	Unimplemented extension
READONLY keyword	Implemented

Table 9 Fortran 77 language extensions —(continued)

Extension	Status
REAL*4	Implemented
REAL*8	Implemented
REAL*16	Implemented
RECORD type	Implemented
RECORDSIZE keyword	Unimplemented extension
RECORDTYPE keyword	Unimplemented extension
TASK COMMON (See the section "TASK COMMON" on page 39)	Implemented
STATIC	Implemented
TYPE keyword	Implemented
TYPE statement	Implemented
Binary data file format conversions	Unimplemented extension
Extended-range DO loops	Implemented
Fortran 90 array notation	Implemented
Hex constants	Implemented
Hollerith constants	Implemented
Integers in logical expressions	Unimplemented extension
List-directed sequential internal I/O	Implemented
Namelist-directed sequential external I/O	Implemented
Octal constants	Implemented
User-defined conversions	Unimplemented extension
Z edit descriptor	Unimplemented extension

Enabling node-parallelism

In the SPP1000-Series compilers (/usr/convex/bin/cc, /usr/convex/bin/fc), node-level parallelism (which is indicated using directives or pragmas) is enabled by default. However, in the Exemplar compilers, loop, task, and region node-parallelism is disabled by default. In other words, `+Ononodepar` is the default.

The `+Ononodepar` option causes the compiler to generate code for a single-node machine. When this option is used, serial code is generated for node-parallel constructs; thus, node-parallelism is not implemented. Thread-parallelism—both automatic and directive-specified—is still implemented.

Use the `+Onodepar` option to enable directive-specified node-parallelism when compiling with `+Oparallel` at `+O3` or `+O4`.

The `+O[no]nodepar` option is only meaningful when specified with the `+Oparallel` option at `+O3` or `+O4`. See the section "`+O[no]nodepar`" on page 19 for more information.

Accessing CPSlib

CPSlib is the Compiler Parallel Support library—a library of low-level parallelization and synchronization routines. In the SPP1000-Series compilers (/usr/convex/bin/cc, /usr/convex/bin/fc), CPSlib is automatically linked in at every optimization level. However, in the Exemplar compilers, CPSlib is automatically linked in only 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 -lpthread -lcps -lpthread -lail prog.c
```

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

See the *Exemplar Programming Guide* for more information on CPSlib routines.

The Exemplar assembler and linker

4

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 assembler `as` differs from the HP-UX `as` in only one way: it supports the `.parallel` directive. This directive sets the parallel attribute in the header of ESOM object files and is provided so that assembly code that was assembled using /usr/convex/bin/as will assemble under the Exemplar assembler. The Exemplar assembler is the default assembler on SPP1200 and SPP1600 systems and on S2000 and X2000 servers.

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-delimited 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 on SPP1200 and SPP1600 systems and on S2000 and X2000 servers. It differs from the standard HP's `ld` in that it:

- Supports thread-local storage
- Supports the memory classes of the Exemplar programming model
- Produces both SOM and ESOM files (the standard HP `ld` produces only SOM files)

Also, the Exemplar linker is required for using the CXdb debugger and the CXpa profiler.

Some of the options that are specific to the Exemplar linker are:

`+parallel`

Sets the parallel flag in the ESOM auxiliary header, enabling parallel execution.

`+tnode` *n*

Sets the maximum number of threads allocated on each node to *n*. The maximum number of threads cannot be less than 1. The default for the maximum number of threads per node is 8 for SPP1200 and SPP1600 systems and 16 for S2000 and X2000 servers.

`+max` *n*

Sets the maximum number of processors needed for the parallel program to *n*. The maximum number of processors cannot be less than the minimum number of processors. If the `+min` is not used, the minimum number of processors is also set to *n*.

`+over`

Sets the oversubscription flag in the executable. Oversubscribing allows the executable to create more than one thread per processor within the subcomplex where the program is running. (See the section "Subcomplexes" on page 71 for information on subcomplexes.) By default this flag is not set.

See the ld(1) man page for more information on these and other Exemplar linker options.

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

SOM vs. ESOM

Two kinds of executable files exist on SPP-UX platforms:

- Standard Object Module (SOM)
- Extended Standard Object Module (ESOM)

The SOM format is used for HP-UX executables. The ESOM format runs only on SPP-UX servers. The ESOM format was derived from the SOM format to support multithreaded processes.

If you link using the Exemplar compiler driver, you will get an ESOM executable. If you use the linker directly without specifying the `+tm` *target* option, the resulting executable will be in the SOM format.

Use the `file` utility or the `chatr` utility to determine if a program is in the SOM or ESOM format. See Chapter 6, "System utilities," or the file(1) and chatr(1) man pages for more information.

Linking to debug or profile

The Exemplar linker, `ld`, is required if you want to use the CXdb debugger (`cxdb`) or the CXpa profiler (`cxpa`). 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. See Chapter 5, "Debugging and profiling," for more information on using these tools.

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
```

Debugging and profiling 5

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
- *CXpa Reference*
- CXpa online help system
- cxpa(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, C, and C++ 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, 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 -Wl,-aarchive_shared
```

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. The `-Wl,-aarchive_shared` option links in archive libraries; if an archive version of a library is not found, the shared version is used. CXdb does not currently support debugging of shared libraries.

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 CXpa profiler

CXpa is a performance analysis tool for monitoring program performance at user-selectable source code regions, such as routines, loops, and compiler-generated parallel 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

Features of CXpa 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 and PVM applications

For more information about these events, see the cxpa(1) man page, the CXpa online help system, or the *CXpa Reference.*

Using CXpa

The basic steps in the profiling process are as follows:

1. Prepare the program for profiling, either by compiling with the `+pa` option or instrumenting it with the `cxoi` utility (refer to the cxoi(1) man page for more information).
2. Link the application with the `+pa` option using the Exemplar linker by means of the compiler driver.
3. Invoke CXpa (`cxpa`), specifying the name of the executable you want to profile.
4. 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.
5. Run the program and generate a performance data file (PDF) by

   Running the executable under the control of CXpa

   or

   writing instrumentation selections to the executable file, exiting CXpa, and then running the executable outside CXpa to generate performance data files for later analysis with CXpa.
6. Analyze the results in graphs and reports.

Note

The `+pa` option is not compatible with `-p` or `-G` options or with the `+O4` or `+Oall` optimization levels.

System utilities 6

This chapter describes various utilities that allow you to make more effective use of your Exemplar servers. Before discussing any of these utilities, however, we need to discuss *subcomplexes* because some of the utilities (such as `mpa`, `scm`, and `sysinfo`) work with subcomplexes.

Subcomplexes

On Exemplar servers, processes run on subcomplexes, which are collections of processors and global memory. Subcomplexes are highly configurable, and configuration is done by the system administrator using the Subcomplex Manager. For more information on the Subcomplex Manager, refer to the scm(1) man page or the *SPP-UX System Administrator's Guide.*

Subcomplexes allow the system administrator to tailor processors and memory to specific application needs, making the most efficient use of system resources. For example, a nonparallel or nontime-critical application can be allocated a single processor; an application containing a lot of fine-grain parallelism can be allocated many processors; and an application requiring large amounts of memory can be allocated processors on several hypernodes. (A hypernode is a set of processors and physical memory organized as a symmetric multiprocessor, or SMP, running a single image of the operating system microkernel.)

Physical configuration

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

Figure 1 shows a 4-hypernode, 64-processor X2000 server split into four subcomplexes.

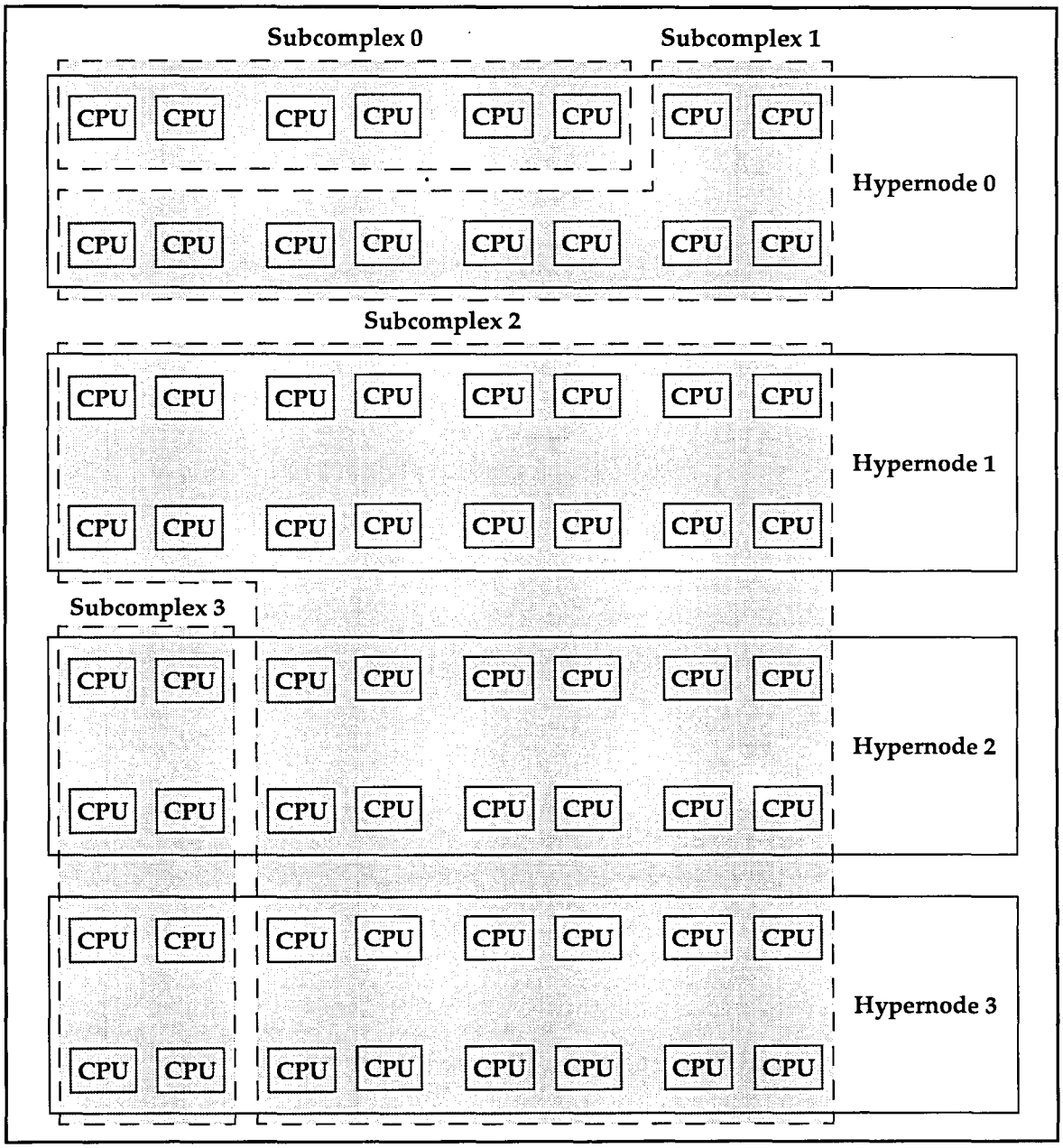


Figure 1 Hypothetical subcomplex configurations

See the *Exemplar Programming Guide* for more information on subcomplexes.

Using the `chatr` utility

The `chatr` (change attributes) utility can be used to print a file's attributes to standard output and to change the file's attributes.

Examples of using `chatr`

The first example shows the `chatr` output (file attributes) on a serial executable:

```
% chatr a.out
a.out:
         ESOM shared executable
         shared library dynamic path search:
             SHLIB_PATH      disabled  second
             embedded path   disabled  first   Not Defined
         internal name:
             a.out
         shared library list:
             dynamic    /usr/lib/libc.1e
         shared library binding:
             deferred
```

The following example shows the `chatr` output from a parallel executable:

```
% chatr parallel-a.out
parallel-a.out:
         ESOM shared executable
         shared library dynamic path search:
             SHLIB_PATH      disabled  second
             embedded path   disabled  first   Not Defined
         internal name:
             parallel-a.out
         shared library list:
             dynamic    /usr/lib/libpthread.1
             dynamic    /usr/lib/libcps.1
             dynamic    /usr/lib/libail.sl
             dynamic    /usr/lib/libc.1e
         shared library binding:
             deferred
```

This output lists the extra libraries linked into a parallel executable.

Getting additional `chatr` information

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

Using the `file` utility

The `file` utility is used to determine file types. It can be used to determine an abundance of information including whether an executable file's parallel flag is set or whether an executable file is in the SOM or ESOM format. (See the section "SOM vs. ESOM" on page 62 for more information on these formats.)

Examples of using `file`

The first example below shows that the a.out is in the ESOM format:

```
% file a.out
a.out: PA-RISC2.0 ESOM shared executable dynamically linked -not stripped
```

The next example shows the output when `file` is used on a parallel a.out:

```
% file a.out
a.out: PA-RISC2.0 ESOM shared executable dynamically linked parallel -not stripped
```

If a file is in the SOM format, `SOM` does not appear in output; the absence of `ESOM` indicates a SOM file, as shown in the following example:

```
% file a.out
a.out: PA-RISC2.0 shared executable dynamically linked -not stripped
```

Getting additional `file` information

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

Using the mpa utility

The mpa utility is used to modify program attributes of an executable file. It can control the:

- Subcomplex that an application runs on
- Size of the data segment and the stack size
- Maximum and minimum numbers of threads needed for parallel execution
- Types of memory used for the user stack and thread-specific memory

Changes to a program's attributes can be made temporarily (for a single run of the executable) or permanently (the executable maintains the attributes until changed) using -m.

Examples of using mpa

This section shows several short examples of how to use the mpa utility. The examples assume that the executable a.out was compiled using +O3 +Oparallel or +O4 +Oparallel, and hence is a parallel executable.

Run the program a.out on the subcomplex named chemistry:

```
% mpa -sc chemistry a.out
```

Run the program a.out using at least 8 threads, but not more than 16 threads:

```
% mpa -min 8 -max 16 a.out
```

Run the program a.out, setting the number of threads the executable will demand to four. If four processors are not available, the -over option enables oversubscription (running more than one thread per processor):

```
% mpa -min 4 -max 4 -over a.out
```

Run the program a.out, setting the stack size to the largest size supported by the system (by specifying -STACK in all uppercase without an argument):

```
% mpa -STACK a.out
```

You can set the stack size to sizes less than the largest allowed by the system by using -stack *s*, where *s* is a positive integer.

Getting additional mpa information

For more information on using mpa, see the mpa(1) man page.

Using the `sysinfo` utility

The `sysinfo` command provides system information regarding memory, processors, and subcomplexes. This command can produce information such as:

- Unix server version string
- Basic processor information, including total number of processors
- Load averages for the system, a certain node, or a certain subcomplex
- Amounts of total, allocated, and free memory

Examples of using `sysinfo`

This section provides examples of how to use the `sysinfo` utility.

Display all available system information:

```
% sysinfo -a
```

Using `-a` shows each individual option (except `-a`) to `sysinfo` followed by its corresponding output. Due to its length, the output from `sysinfo -a` is not shown.

Display the total number of CPUs in the system:

```
% sysinfo -cpu_count
16
```

The output indicates that the current system has 16 processors.

Display the load average for all subcomplexes:

```
% sysinfo -ls
System     load average:          1.86   1.92   2.02
EPM        load average:          0.58   0.69   0.83
MLIB       load average:          1.07   1.04   0.79
```

The output shows three subcomplexes (`System`, `EPM`, and `MLIB`) and their respective load averages.

Display memory statistics for the entire system:

```
% sysinfo -memc
COMPLEX MEMORY     max    allocated     free
global              0M           0M       0M
node private     1843M         783M    1060M
buffer cache      204M
network cache       0M
total            2048M         783M    1060M
```

The `sysinfo -memc` output provides information on the various types of memory available.

Getting additional `sysinfo` information

For more information on using `sysinfo`, see the sysinfo(1) man page.

Additional utilities

Table 10 lists other system utilities that you may find helpful in using your Exemplar server. Sources for additional information on these utilities are given in the second column of the table.

Table 10 Additional system utilities

`cnx_ps`	Prints thread information for processes. See the cnx_ps(1) man page for more information.
`make`	Maintains, updates, and regenerates groups of programs. See the make(1) man page for more information.
`mnm`	Monitors total memory usage per hypernode; use `-h` to get a listing of options.
`nm`	Prints the name list of an object file or library; `nm` is useful to see where symbols are defined and referenced. See the nm(1) man page for more information.
`pot`	The `pot` utility is similar to the `top` utility but is thread-based rather than process-based. (`top` displays information about the top processes—in terms of CPU usage—on a system.) `pot` can display various information; the particular information displayed and the order in which it is displayed can be configured by the user. Use `top` for gathering data on serial processes and `pot` for data on serial/parallel processes. See the pot(1) man page for more information.
`scm`	Subcomplex Manager: shows configuration of processors and memory resources. See the scm(1) man page for more information.
`size`	Prints section sizes of object files—text, data, bss (uninitialized data), and the total size. This information is helpful in determining if your program is within the tunables limits. (Tunables are specified in the file /stand/tunables.) See the size(1) man page for more information.
`sod`	Displays standard format object files in a human-readable form; using the `-a` option displays all of the auxiliary headers—these headers contain information about the minimum and maximum CPUs and the memory types. See the sod(1) man page for more information.
`syspic`	Monitors system performance. See the syspic(1) man page for more information.
`top`	Displays and updates information about the top processes on the system (processes are ranked by raw CPU percentage). Use `top` for gathering data on serial processes and `pot` for data on serial/parallel processes. See the top(1) man page for more information.

Environment variables

A

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`

Note

The `MP_NUMBER_OF_THREADS` environment variable is the only environment variable starting with `MP_` that is accepted by the Exemplar compilers using either `+Okernel_threads` or `+Oprocess_threads`. Other environment variables starting with `MP_` are accepted only if `+Oprocess_threads` is specified.

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-delimited 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-delimited 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.

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 (in the subcomplexes where the programs run) that are to execute programs that have been compiled, using `+Oparallel`, for parallel execution. If not set, it defaults to the number of processors on the executing subcomplex. This variable is used at runtime by parallel programs compiled using either the C or the Fortran 77 compiler.

Setting this environment variable to a value of *n* is the same as using "`mpa -max` *n*." However, the `mpa` utility can be used to set the minimum, as well as the maximum, number of threads used when executing a parallel program. See the mpa(1) man page for more information.

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

altdir

is the name of the alternative directory for temporary files

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

X

B5600-90002