PGI® Compiler User's Guide

Parallel Fortran, C and C++ for Scientists and Engineers

Release 2013

The Portland Group®

PGI® Compiler User's Guide

Printed in the United States of America
First Printing: Release 2012, 12.1, January 2012
Second Printing: Release 2012, 12.6, June 2012
Third Printing: Release 2012, 12.9, September 2012
Fourth Printing: Release 2013, 13.1, January 2013
Fifth Printing: Release 2013, 13.2, February 2013
Sixth Printing: Release 2013, 13.3, March 2013
Seventh Printing: Release 2013, 13.6, June 2013
Eighth Printing: Release 2013, 13.8, August 2013
Ninth Printing: Release 2013, 13.9, September 2013
Tenth Printing: Release 2013, 13.10, October 2013

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

ID: 132831041

Contents

Tables

Examples

Preface

This guide is part of a set of manuals that describe how to use The Portland Group (PGI) Fortran, C, and C++ compilers and program development tools. These compilers and tools include the *PGF77*, *PGF95*, PGFORTRAN, *PGC++*, and *PGCC ANSI* C compilers, the *PGPROF* profiler, and the *PGDBG* debugger. They work in conjunction with an x86 or x64 assembler and linker. You can use the PGI compilers and tools to compile, debug, optimize, and profile serial and parallel applications for x86 processor-based systems.

The *PGI Compiler User's Guide* provides operating instructions for the PGI command-level development environment. The *PGI Compiler Reference Manual* contains details concerning the PGI compilers' interpretation of the Fortran language, implementation of Fortran language extensions, and command-level compilation. Users are expected to have previous experience with or knowledge of the Fortran programming language.

Audience Description

This manual is intended for scientists and engineers using the PGI compilers. To use these compilers, you should be aware of the role of high-level languages, such as Fortran, C, and C++, as well as assembly-language in the software development process; and you should have some level of understanding of programming. The PGI compilers are available on a variety of x86 or x64 hardware platforms and operating systems. You need to be familiar with the basic commands available on your system.

Compatibility and Conformance to Standards

Your system needs to be running a properly installed and configured version of the compilers. For information on installing PGI compilers and tools, refer to the Release Notes and Installation Guide included with your software.

For further information, refer to the following:

- *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).
- *ISO/IEC 1539-1 : 1991, Information technology – Programming Languages – Fortran,* Geneva, 1991 (Fortran 90).
- *ISO/IEC 1539-1 : 1997, Information technology – Programming Languages – Fortran,* Geneva, 1997 (Fortran 95).

- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *OpenMP Application Program Interface*, Version 2.5, May 2005, http://www.openmp.org.
- *Programming in VAX Fortran*, Version 4.0, Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- *American National Standard Programming Language C*, ANSI X3.159-1989.
- ISO/IEC 9899:1999, Information technology – Programming Languages – C, Geneva, 1999 (C99).

Organization

Users typically begin by wanting to know how to use a product and often then find that they need more information and facts about specific areas of the product. Knowing *how* as well as *why* you might use certain options or perform certain tasks is key to using the PGI compilers and tools effectively and efficiently. However, once you have this knowledge and understanding, you very likely might find yourself wanting to know much more about specific areas or specific topics.

This book contains the essential information on how to use the compiler and is divided into these chapters:

Chapter 1, "*Getting Started*" provides an introduction to the PGI compilers and describes their use and overall features.

Chapter 2, "*Using Command Line Options*" provides an overview of the command-line options as well as task-related lists of options.

Chapter 3, "*Optimizing & Parallelizing*" describes standard optimization techniques that, with little effort, allow users to significantly improve the performance of programs.

Chapter 4, "*Using Function Inlining*" describes how to use function inlining and shows how to create an inline library.

Chapter 5, "*Using OpenMP*" provides a description of the OpenMP Fortran parallelization directives and of the OpenMP C and C++ parallelization pragmas and shows examples of their use.

Chapter 6, "*Using MPI* " describes how to use MPI with PGI Workstation and PGI server.

Chapter 7, "*Using an Accelerator*" describes how to use the PGI Accelerator compilers.

Chapter 8, "*Eclipse*" describes Eclipse, a free, open source, integrated software development environment.

Chapter 9, "*Using Directives and Pragmas*" provides a description of each Fortran optimization directive and C/C++ optimization pragma, and shows examples of their use.

Chapter 10, "*Creating and Using Libraries*" discusses PGI support libraries, shared object files, and environment variables that affect the behavior of the PGI compilers.

Chapter 11, " *Using Environment Variables*" describes the environment variables that affect the behavior of the PGI compilers.

Chapter 12, "*Distributing Files - Deployment*" describes the deployment of your files once you have built, debugged and compiled them successfully.

Chapter 13, "*Inter-language Calling*" provides examples showing how to place C Language calls in a Fortran program and Fortran Language calls in a C program.

Chapter 14, "*Programming Considerations for 64-Bit Environments*" discusses issues of which programmers should be aware when targeting 64-bit processors.

Chapter 15, "*C/C++ Inline Assembly and Intrinsics*" describes how to use inline assembly code in C and C++ programs, as well as how to use intrinsic functions that map directly to x86 and x64 machine instructions.

Hardware and Software Constraints

This guide describes versions of the PGI compilers that produce assembly code for x86 and x64 processor-based systems. Details concerning environment-specific values and defaults and system-specific features or limitations are presented in the release notes delivered with the PGI compilers.

Conventions

The *PGI Compiler User's Guide* uses the following conventions:

italic
: Italic font is for emphasis.

`Constant Width`
: Constant width font is for commands, filenames, directories, examples and for language statements in the text, including assembly language statements.

[item1]
: Square brackets indicate optional items. In this case item1 is optional.

{ item2 | item 3}
: Braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename...
: Ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN
: Fortran language statements are shown in the text of this guide using upper-case characters and a reduced point size.

The PGI compilers and tools are supported on both 32-bit and 64-bit variants of Linux, Mac OS X, and Windows operating systems on a variety of x86-compatible processors. There are a wide variety of releases and distributions of each of these types of operating systems. Further, The PGI *Compiler User's Guide* uses a number of terms with respect to these platforms. For a complete definition of these terms and other terms in this guide with which you may be unfamiliar, PGI provides a glossary of terms which you can access at *www.pgroup.com/support/definitions.htm*.

AMD64	linux86	osx86	static linking
barcelona	linux86-64	osx86-64	Win32
DLL	Mac OS X	shared library	Win64
driver	-mcmodel=small	SSE	Windows
dynamic library	-mcmodel=medium	SSE1	x64
EM64T	MPI	SSE2	x86
hyperthreading (HT)	MPICH	SSE3	x87
IA32	multi-core	SSE4A and ABM	
Large arrays	NUMA	SSSE3	

The following table lists the PGI compilers and tools and their corresponding commands:

Table 1. PGI Compilers and Commands

Compiler or Tool	Language or Function	Command
PGF77	FORTRAN 77	pgf77
PGF95	Fortran 90/95/F2003	pgf95
PGFORTRAN	PGI Fortran	pgfortran
PGCC C	*ANSI* C99 and K&R C	pgcc
PGC++	*ANSI* C++ with cfront features	pgcpp on Windows pgCC and pgcpp on Linux
PGDBG	Source code debugger	pgdbg
PGPROF	Performance profiler	pgprof

Note

The commands **pgf95** and **pgfortran** are equivalent.

In general, the designation *PGI Fortran* is used to refer to The Portland Group's Fortran 90/95/F2003 compiler, and *pgfortran* is used to refer to the command that invokes the compiler. A similar convention is used for each of the PGI compilers and tools.

For simplicity, examples of command-line invocation of the compilers generally reference the *pgfortran* command, and most source code examples are written in Fortran. Usage of the *PGF77* compiler, whose features are a subset of *PGF95* or *PGFORTRAN*, is similar. Usage of *PGC++* and *PGCC* is consistent with *PGF95, PGFORTRAN*, and *PGF77*, though there are command-line options and features of these compilers that do not apply to *PGF95, PGFORTRAN*, and *PGF77*, and vice versa.

There are a wide variety of x86-compatible processors in use. All are supported by the PGI compilers and tools. Most of these processors are forward-compatible, but not backward-compatible, meaning that code compiled to target a given processor will not necessarily execute correctly on a previous-generation processor.

A table listing the processor options that PGI supports is available in the Release Notes. The table also includes the features utilized by the PGI compilers that distinguish them from a compatibility standpoint.

In this manual, the convention is to use "x86" to specify the group of processors that are "32-bit" but not "64-bit." The convention is to use "x64" to specify the group of processors that are both "32-bit" and "64-bit." x86 processor-based systems can run only 32-bit operating systems. x64 processor-based systems can run either 32-bit or 64-bit operating systems, and can execute all 32-bit x86 binaries in either case. x64 processors have additional registers and 64-bit addressing capabilities that are utilized by the PGI compilers and tools when running on a 64-bit operating system. The prefetch, SSE1, SSE2 and SSE3 processor features further distinguish the various processors. Where such distinctions are important with respect to a given compiler option or feature, it is explicitly noted in this manual.

Note

The default for performing scalar floating-point arithmetic is to use SSE instructions on targets that support SSE1 and SSE2.

Related Publications

The following documents contain additional information related to the x86 and x64 architectures, and the compilers and tools available from The Portland Group.

- *PGI Fortran Reference* manual describes the F2003, FORTRAN 77, and Fortran 90/95 statements, data types, input/output format specifiers, and additional reference material related to use of the PGI Fortran compilers.
- *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- *System V Application Binary Interface X86-64 Architecture Processor Supplement*, *www.x86-64.org/abi.pdf*.
- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *Programming in VAX Fortran, Version 4.0,* Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran,* IBM Corporation, Rev. GC26-4119.
- *The C Programming Language* by Kernighan and Ritchie (Prentice Hall).
- *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990).
- *OpenMP Application Program Interface*, Version 2.5 May 2005 (OpenMP Architecture Review Board, 1997-2005).

Chapter 1. Getting Started

This chapter describes how to use the PGI compilers.

The command used to invoke a compiler, such as the pgfortran command, is called a *compiler driver*. The compiler driver controls the following phases of compilation: preprocessing, compiling, assembling, and linking. Once a file is compiled and an executable file is produced, you can execute, debug, or profile the program on your system. Executables produced by the PGI compilers are unconstrained, meaning they can be executed on any compatible x86 or x64 processor-based system, regardless of whether the PGI compilers are installed on that system.

Overview

In general, using a PGI compiler involves three steps:

1. Produce program source code in a file containing a `.f` extension or another appropriate extension, as described in "Input Files," on page 4. This program may be one that you have written or one that you are modifying.
2. Compile the program using the appropriate compiler command.
3. Execute, debug, or profile the executable file on your system.

You might also want to deploy your application, though this is not a required step.

The PGI compilers allow many variations on these general program development steps. These variations include the following:

- Stop the compilation after preprocessing, compiling or assembling to save and examine intermediate results.
- Provide options to the driver that control compiler optimization or that specify various features or limitations.
- Include as input intermediate files such as preprocessor output, compiler output, or assembler output.

Invoking the Command-level PGI Compilers

To translate and link a Fortran, C, or C++ program, the pgf77, pgf95, pgfortran, pgcc, pgcpp, and pgc++ commands do the following:

1. Preprocess the source text file.
2. Check the syntax of the source text.
3. Generate an assembly language file.
4. Pass control to the subsequent assembly and linking steps.

Example 1.1. Hello program

Let's look at a simple example of using the PGI compiler to create, compile, and execute a program that prints *hello*.

Step 1: Create your program.

For this example, suppose you enter the following simple Fortran program in the file `hello.f`:

```
print *, "hello"
 end
```

Step 2: Compile the program.

When you created your program, you called it `hello.f`. In this example, we compile it from a shell command prompt using the default pgfortran driver option. Use the following syntax:

```
PGI$ pgfortran hello.f
PGI$
```

By default, the executable output is placed in the file `a.out`, or, on Windows platforms, in a filename based on the name of the first source or object file on the command line. However, you can specify an output file name by using the `-o` option.

To place the executable output in the file `hello`, use this command:

```
PGI$ pgfortran -o hello hello.f
PGI$
```

Step 3: Execute the program.

To execute the resulting hello program, simply type the filename at the command prompt and press the Return or Enter key on your keyboard:

```
PGI$ hello
 hello
PGI$
```

Command-line Syntax

The compiler command-line syntax, using pgfortran as an example, is:

```
pgfortran [options] [path]filename [...]
```

Where:

options
: is one or more command-line options, all of which are described in detail in Chapter 2, "*Using Command Line Options*".

path
: is the pathname to the directory containing the file named by filename. If you do not specify the path for a filename, the compiler uses the current directory. You must specify the path separately for each filename not in the current directory.

filename
: is the name of a source file, preprocessed source file, assembly-language file, object file, or library to be processed by the compilation system. You can specify more than one [path]filename.

Command-line Options

The command-line options control various aspects of the compilation process. For a complete alphabetical listing and a description of all the command-line options, refer to Chapter 2, "*Using Command Line Options*".

The following list provides important information about proper use of command-line options.

- Case is significant for options and their arguments.
- The compiler drivers recognize characters preceded by a hyphen (-) as command-line options. For example, the `-Mlist` option specifies that the compiler creates a listing file.

  ### Note

  The convention for the text of this manual is to show command-line options using a dash instead of a hyphen; for example, you see `-Mlist`.

- The PGC++ command recognizes a group of characters preceded by a plus sign (+) as command-line options.
- The order of options and the filename is flexible. That is, you can place options before and after the filename argument on the command line. However, the placement of some options is significant, such as the `-l` option, in which the order of the filenames determines the search order.

  ### Note

  If two or more options contradict each other, the *last* one in the command line takes precedence.

Fortran Directives and C/C++ Pragmas

You can insert Fortran directives and C/C++ pragmas in program source code to alter the effects of certain command-line options and to control various aspects of the compilation process for a specific routine or a specific program loop. For more information on Fortran directives and C/C++ pragmas, refer to Chapter 5, "*Using OpenMP*" and Chapter 9, "*Using Directives and Pragmas*".

Filename Conventions

The PGI compilers use the filenames that you specify on the command line to find and to create input and output files. This section describes the input and output filename conventions for the phases of the compilation process.

Input Files

You can specify assembly-language files, preprocessed source files, Fortran/C/C++ source files, object files, and libraries as inputs on the command line. The compiler driver determines the type of each input file by examining the filename extensions.

Note

For systems with a case-insensitive file system, use the `-Mpreprocess` option, described in "Command-Line Options Reference" chapter of the PGI Compiler Reference Manual, under the commands for Fortran preprocessing.

The drivers use the following conventions:

filename.f
: indicates a Fortran source file.

filename.F
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.FOR
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.F95
: indicates a Fortran 90/95 source file that can contain macros and preprocessor directives (to be preprocessed).

filename.f90
: indicates a Fortran 90/95 source file that is in freeform format.

filename.f95
: indicates a Fortran 90/95 source file that is in freeform format.

filename.cuf
: indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions.

filename.CUF
: indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions and that can contain macros and preprocessor directives (to be preprocessed).

filename.c
: indicates a C source file that can contain macros and preprocessor directives (to be preprocessed).

filename.C
: indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.i
: indicates a preprocessed C or C++ source file.

filename.cc
: indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.cpp
: indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.s
: indicates an assembly-language file.

filename.o
: (Linux and Mac OS X) indicates an object file.

filename.obj
: (Windows systems only) indicates an object file.

filename.a
: (Linux and Mac OS X) indicates a library of object files.

filename.lib
: (Windows systems only) indicates a statically-linked library of object files or an import library.

filename.so
: (Linux only) indicates a library of shared object files.

filename.dll
: (Windows systems only) indicates a dynamically-linked library.

filename.dylib
: (Mac OS X systems only) indicates a dynamically-linked library.

The driver passes files with `.s` extensions to the assembler and files with .o, .obj, .so, .dll, .a and .lib extensions to the linker. Input files with unrecognized extensions, or no extension, are also passed to the linker.

Files with a `.F` (Capital F) or `.FOR` suffix are first preprocessed by the Fortran compilers and the output is passed to the compilation phase. The Fortran preprocessor functions like cpp for C/C++ programs, but is built in to the Fortran compilers rather than implemented through an invocation of cpp. This design ensures consistency in the preprocessing step regardless of the type or revision of operating system under which you're compiling.

Any input files not needed for a particular phase of processing are not processed. For example, if on the command line you specify an assembly-language file (filename.s) and the `-S` option to stop before the assembly phase, the compiler takes no action on the assembly language file. Processing stops after compilation and the assembler does not run. In this scenario, the compilation must have been completed in a previous pass which created the `.s` file. For a complete description of the `-S` option, refer to the following section:"Output Files".

In addition to specifying primary input files on the command line, code within other files can be compiled as part of include files using the `INCLUDE` statement in a Fortran source file or the `preprocessor #include` directive in Fortran source files that use a `.F` extension or C and C++ source files.

When linking a program with a library, the linker extracts only those library components that the program needs. The compiler drivers link in several libraries by default. For more information about libraries, refer to Chapter 10, "*Creating and Using Libraries*".

Output Files

By default, an executable output file produced by one of the PGI compilers is placed in the file a.out, or, on Windows, in a filename based on the name of the first source or object file on the command line. As the example in the preceding section shows, you can use the `-o` option to specify the output file name.

If you use one of the options: `-F` (Fortran only), `-P` (C/C++ only), `-S` or `-c`, the compiler produces a file containing the output of the last completed phase for each input file, as specified by the option supplied. The output file will be a preprocessed source file, an assembly-language file, or an unlinked object file respectively. Similarly, the `-E` option does not produce a file, but displays the preprocessed source file on the standard output. Using any of these options, the `-o` option is valid only if you specify a single input file. If no errors occur during processing, you can use the files created by these options as input to a future invocation of any of the PGI compiler drivers. The following table lists the stop-after options and the output files that the compilers create when you use these options. It also describes the accepted input files.

Table 1.1. Stop-after Options, Inputs and Outputs

Option	Stop after	Input	Output
–E	preprocessing	Source files.	preprocessed file to standard out
–F	preprocessing	Source files. This option is not valid for pgcc or pgcpp.	preprocessed file (.f)
–P	preprocessing	Source files. This option is not valid for pgf77, pgf95, or pgfortran.	preprocessed file (.i)
–S	compilation	Source files or preprocessed files.	assembly-language file (.s)
–c	assembly	Source files, preprocessed files or assembly-language files.	unlinked object file (.o or .obj)
none	linking	Source files, preprocessed files, assembly-language files, object files or libraries.	executable file (a.out or .exe)

If you specify multiple input files or do not specify an object filename, the compiler uses the input filenames to derive corresponding default output filenames of the following form, where filename is the input filename without its extension:

filename.f
: indicates a preprocessed file, if you compiled a Fortran file using the `-F` option.

filename.i
: indicates a preprocessed file, if you compiled using the `-P` option.

filename.lst
: indicates a listing file from the `-Mlist` option.

filename.o or filename.obj
: indicates an object file from the `-c` option.

filename.s
: indicates an assembly-language file from the `-S` option.

> **Note**
>
> Unless you specify otherwise, the destination directory for any output file is the current working directory. If the file exists in the destination directory, the compiler overwrites it.

The following example demonstrates the use of output filename extensions.

```
$ pgfortran -c proto.f proto1.F
```

This produces the output files proto.o and proto1.o, or, on Windows, proto.obj and proto1.obj all of which are binary object files. Prior to compilation, the file proto1.F is preprocessed because it has a .F filename extension.

Fortran, C, and C++ Data Types

The PGI Fortran, C, and C++ compilers recognize scalar and aggregate data types. A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. An aggregate data type consists of one or more scalar data type objects, such as an array of integer values.

For information about the format and alignment of each data type in memory, and the range of values each type can have on x86 or x64 processor-based systems running a 32-bit operating system, refer to "Fortran, C, and C ++ Data Types" chapter of the PGI Compiler Reference Manual.

For more information on x86-specific data representation, refer to the *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).

For more information on x64 processor-based systems and the application binary interface (ABI) for those systems, see *www.x86-64.org/documentation/abi.pdf*.

Parallel Programming Using the PGI Compilers

The PGI compilers support many styles of parallel programming, such as these:

- Automatic shared-memory parallel programs compiled using the `-Mconcur` option to pgf77, pgf95, pgfortran, pgcc, or pgcpp — parallel programs of this variety can be run on shared-memory parallel (SMP) systems such as dual-core or multi-processor workstations.
- OpenMP shared-memory parallel programs compiled using the `-mp` option to pgf77, pgf95, pgfortran, pgcc, or pgcpp — parallel programs of this variety can be run on SMP systems. Carefully coded user-directed parallel programs using OpenMP directives can often achieve significant speed-ups on dual-core workstations or large numbers of processors on SMP server systems. Chapter 5, "*Using OpenMP*" contains complete descriptions of user-directed parallel programming.
- Distributed computing using an MPI message-passing library for communication between distributed processes.

- Accelerated computing using either a low-level model such as CUDA Fortran or a high-level model such as the PGI Accelerator model or OpenACC to target a many-core GPU or other attached accelerator.

In this manual, the first two types of parallel programs are collectively referred to as SMP parallel programs.

On a single silicon die, today's CPUs incorporate two or more complete processor cores - functional units, registers, level 1 cache, level 2 cache, and so on. These CPUs are known as multi-core processors. For purposes of threads or OpenMP parallelism, these cores function as two or more distinct processors. However, the processing cores are on a single chip occupying a single socket on a system motherboard. For purposes of PGI software licensing, a multi-core processor is treated as a single CPU.

Running SMP Parallel Programs

When you execute an SMP parallel program, by default it uses only one processor. To run on more than one processor, set the `NCPUS` environment variable to the desired number of processors. For information on how to set environment variables, refer to"Setting Environment Variables," on page 135

Note

If you set NCPUS to a number larger than the number of physical processors, your program may execute very slowly.

Platform-specific considerations

The following list are the platforms supported by the PGI Workstation and PGI Server compilers and tools:

- 32-bit Linux – supported on 32-bit Linux operating systems running on either a 32-bit x86 compatible or an x64 compatible processor.
- 4-bit/32-bit Linux – includes all features and capabilities of the 32-bit Linux version, and is also supported on 64-bit Linux operating systems running on an x64 compatible processor.
- 32-bit Windows – supported on 32-bit Windows operating systems running on either a 32-bit x86 compatible or an x64-compatible processor.
- 64-bit/32-bit Windows – includes all features and capabilities of the 32-bit Windows version; also supported on 64-bit Windows operating systems running an x64- compatible processor.
- 32-bit Mac OS X – supported on 32-bit Mac OS X operating systems running on either a 32-bit or 64-bit Intel-based Mac system.
- 64-bit Mac OS X – supported on 64-bit Mac OS X operating systems running on a 64-bit Intel-based Mac system.

The following sections describe the specific considerations required to use the PGI compilers on the various platforms: Linux, Windows, and Mac OS X.

Using the PGI Compilers on Linux

Linux Header Files

The Linux system header files contain many GNU gcc extensions. PGI supports many of these extensions, thus allowing the PGCC C and C++ compilers to compile most programs that the GNU compilers can compile. A few header files not interoperable with the PGI compilers have been rewritten. These files are included in `$PGI/linux86/include`, such as `sigset.h`, `asm/byteorder.h`, `stddef.h`, `asm/posix_types.h` and others. Also, PGI's version of `stdarg.h` supports changes in newer versions of Linux.

If you are using the PGCC C or C++ compilers, please make sure that the supplied versions of these include files are found before the system versions. This hierarchy happens by default unless you explicitly add a –I option that references one of the system `include` directories.

Running Parallel Programs on Linux

You may encounter difficulties running auto-parallel or OpenMP programs on Linux systems when the per-thread stack size is set to the default (2MB). If you have unexplained failures, please try setting the environment variable `OMP_STACKSIZE` to a larger value, such as 8MB. For information on setting environment variables, refer to "Setting Environment Variables," on page 135.

If your program is still failing, you may be encountering the hard 8 MB limit on main process stack sizes in Linux. You can work around the problem by issuing the following command in csh:

```
% limit stacksize unlimited
```

in bash, sh, zsh, or ksh, use:

```
$ ulimit -s unlimited
```

Using the PGI Compilers on Windows

PGI on the Windows Start Menu

PGI provides a Start menu entry that provides access to different versions of PGI command shells as well as easy access to the PGI Debugger, the PGI Profiler, documentation, and licensing. The following sections provide a quick overview of the menu selections.

To access the main PGI menu, from the Start menu, select *Start* | *All Programs* | *PGI Workstation*.

Command Shell Submenus

From the PGI Workstation menu, you have access to PGI command shells for each version of PGI installed on your system. For example, if you have both PGI 13.10 and PGI 11.9 installed, then you have a submenu for each of these versions.

The PGI submenus for each version include the following:

- **PGI Bash (64)** – Select this option to launch a Cygwin **bash** shell in which the environment is pre-initialized to use the 64-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on x64 systems with Cygwin installed.)

- **PGI Bash –** Select this option to launch a Cygwin **bash** shell in which the environment is pre-initialized to use the 32-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on systems with Cygwin installed.)
- **PGI Cmd (64) –** Select this option to launch a Microsoft command shell in which the environment is pre-initialized to use the 64-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on x64 systems.)
- **PGI Cmd –** Select this option to launch a Microsoft command shell in which the environment is pre-initialized to use the 32-bit PGI compilers and tools. The default environment variables are already set and available.

The command window launched by PGI Workstation can be customized using the "Properties" selection on the menu accessible by right-clicking the window's title bar.

Debugger & Profiler Submenu

From the Debugger & Profiler menu, you have access to the PGI debugging and profiling tools. PGDBG is a symbolic debugger for Fortran, C, C++ and assembly language programs. It provides debugger features, such as execution control using breakpoints, single-stepping, and examination and modification of application variables, memory locations, and registers.

- **PGDBG Debugger –** Select this option to launch the PGI debugger, PGDBG, for use with both 32-bit and 64-bit applications.
- **PGPROF Performance Profiler –** Select this option to launch the PGPROF Performance Profiler. PGPROF provides a way to visualize and diagnose the performance of the components of your program, and provides features for helping you to understand why certain parts of your program have high execution times.

Documentation Submenu

From the Documentation menu, you have access to all PGI documentation that is useful for PGI users. The documentation that is available includes the following:

- **AMD Core Math Library–** Select this option to display documentation that describes elements of the AMD Core Math Library, a software development library released by AMD that includes a set of useful mathematical routines optimized for AMD processors.
- **CUDA Fortran Reference–** Select this option to display the CUDA Fortran Programming Guide and Reference. This document describes CUDA Fortran, a small set of extensions to Fortran that support and build upon the CUDA computing architecture.
- **Fortran Language Reference–** Select this option to display the *PGI Fortran Reference*. This document describes The Portland Group's implementation of the FORTRAN 77 and Fortran 90/95 languages and presents the Fortran language statements, intrinsics, and extension directives.
- **Installation Guide–** Select this option to display the *PGI Server and Workstation Installation Guide*. This document provides an overview of the steps required to successfully install and license PGI Server and PGI Workstation.
- **PGDBG Debugger Guide–** Select this option to display the *PGDBG Debugger Guide*. This guide describes how to use the PGDBG debugger to debug serial and parallel applications built with PGI compilers. It

contains information about how to use PGDBG, as well as detailed reference information on commands and graphical interfaces.

- **PGPROF Profiler Guide**– Select this option to display the *PGPROF Profiler Guide*. This guide describes how to use the PGPROF profiler to tune serial and parallel applications built with PGI compilers. It contains information about how to use the profiler, as well as detailed reference information on commands and graphical interfaces.
- **Release Notes**– Select this option to display the latest *PGI Server and Workstation Release Notes*. This document describes changes between previous releases and the current release.
- **User's Guide**– Select this option to display the *PGI User's Guide*. This document provides operating instructions for the PGI command-level development environment as well as details concerning the PGI compilers' interpretation of the Fortran language, implementation of Fortran language extensions, and command-level compilation.

Licensing Submenu

From the Licensing menu, you have access to the PGI License Agreement and an automated license generating tool:

- **Generate License**– Select this option to display the PGI License Setup dialog that walks you through the steps required to download and install a license for PGI Workstation or PGI Server. To complete this process you need an internet connection.
- **License Agreement**– Select this option to display the license agreement that is associated with use of PGI software.

PGI on the Windows Desktop

By default, a PGI Workstation installation creates a shortcut on the Windows desktop. This shortcut launches a Cygwin **bash** shell if Cygwin is installed; otherwise it launches a Microsoft command shell. The environment for this shell is pre-configured to use PGI compilers and tools. On 64-bit systems, the 64-bit compilers are targeted, while on 32-bit systems, the 32-bit compilers are targeted.

BASH Shell Environment (Cygwin)

A UNIX-like shell environment, Cygwin, is bundled with PGI compilers and tools for Windows to provide a familiar development environment for Linux or UNIX users.

After installation of PGI Workstation or PGI Server, you have a PGI Workstation icon on your Windows desktop. Double-left-click on this icon to launch an instance of the Cygwin **bash** command shell window. Working within BASH is very much like working within the sh or ksh shells on a Linux system; yet BASH has a command history feature similar to csh and several other unique features. Shell programming is fully supported.

The BASH shell window is pre-initialized for usage of the PGI compilers and tools, so there is no need to set environment variables or modify your command path when the command window comes up. In addition to the PGI compiler commands, within BASH you have access to over 100 common commands and utilities, including but not limited to the following:

vi	gzip / gunzip	ftp
tar / untar	grep / egrep / fgrep	awk
sed	cksum	cp
cat	diff	du
date	kill	ls
find	mv	printenv / env
more / less	touch	wc
rm / rmdir	make	

If you are familiar with program development in a Linux environment, editing, compiling, and executing programs within **bash** will be very comfortable. If you have not previously used such an environment, you might want to familiarize yourself with *vi* or other editors and with `makefiles`. The Web has an extensive online tutorial available for the *vi* editor as well as a number of thorough introductions to the construction and use of `makefiles`.

ar or ranlib

For library compatibility, PGI provides versions of **ar** and **ranlib** that are compatible with native Windows object-file formats. For more information on these commands, refer to "Creating and Using Static Libraries on Windows," on page 123.

Using the PGI Compilers on Mac OS X

PGI Workstation 13.10 for Mac OS X supports most of the features of the 32-and 64-bit versions for linux86 and linux86-64 environments. Typically the PGI compilers and tools on Mac OS X function identically to their Linux counterparts.

Mac OS X Header Files

The Mac OS X header files contain numerous non-standard extensions. PGI supports many of these extensions, thus allowing the PGCC C and C++ compilers to compile most programs that the GNU compilers can compile. A few header files not interoperable with the PGI compilers have been rewritten. These files are included in $PGI/osx86/13.10/include or $PGI/osx86-64/13.10/include. These files are: stdarg.h, stddef.h, and others.

If you are using the PGCC C or C++ compilers, please make sure that the supplied versions of these include files are found before the system versions. This will happen by default unless you explicitly add a –I option that references one of the system include directories.

Mac OS Debugging Requirements

Both the `-g` and `-Mkeepobj` switches play important roles when compiling a program on Apple Mac OS for debugging.

- To debug a program with symbol information on the Mac OS, files must be compiled with the -g switch to keep the program's object files, the files with a ".o" extension. Further, these object files must remain in the same directory in which they were created.

- If a program is built with separate compile and link steps, by compiling with the -c switch which generates the ".o" object files, then using the –g switch guarantees the required object files are available for debugging.

Use the following command sequence to compile and then link your code.

To compile the programs, use these commands:

```
pgcc -c -g main.cpgcc -c -g foo.cpgcc -c -g bar.c
```

To link, use this command:

```
pgcc -g main.o foo.o bar.o
```

Linking on Mac OS X

On the Mac OS X, the PGI Workstation 13.10 compilers do not support static linking of user binaries. For compatibility with future Apple updates, the compilers support dynamic linking of user binaries. For more information on dynamic linking, refer to "Creating and Using Dynamic Libraries on Mac OS X," on page 122.

Running Parallel Programs on Mac OS X

You may encounter difficulties running auto-parallel or OpenMP programs on Mac OS X systems when the per-thread stack size is set to the default (8MB). If you have unexplained failures, please try setting the environment variable OMP_STACKSIZE to a larger value, such as 16MB. For information on how to set environment variables, refer to "Setting Environment Variables," on page 135.

Site-specific Customization of the Compilers

If you are using the PGI compilers and want all your users to have access to specific libraries or other files, there are special files that allow you to customize the compilers for your site.

Using siterc Files

The PGI compiler drivers utilize a file named `siterc` to enable site-specific customization of the behavior of the PGI compilers. The `siterc` file is located in the `bin` subdirectory of the PGI installation directory. Using `siterc`, you can control how the compiler drivers invoke the various components in the compilation tool chain.

Using User rc Files

In addition to the siterc file, user `rc` files can reside in a given user's home directory, as specified by the user's HOME environment variable. You can use these files to control the respective PGI compilers. All of these files are optional.

On Linux and Mac OS X, these files are named `.mypgf77rc`, `.mypgf90rc`, `.mypgccrc`, and `.mypgcpprc`.

On Windows, these files are named `mypgf77rc, mypgf90rc, mypgf95rc, mypgfortranrc, mypgccrc, mypgcpprc`.

The following examples show how these `rc` files can be used to tailor a given installation for a particular purpose.

Table 1.2. Examples of Using siterc and User rc Files

To do this...	Add the line shown to the indicated file
Make available to all linux86-64 compilations the libraries found in `/opt/newlibs/64`	`set SITELIB=/opt/newlibs/64;` to `/opt/pgi/linux86-64/13.10/bin/siterc`
Make available to all linux86 compilations the libraries found in `/opt/newlibs/32`	`set SITELIB=/opt/newlibs/32;` to `/opt/pgi/linux86/13.10/bin/siterc`
Add to all linux86-64 compilations a new library path: `/opt/local/fast`	`append SITELIB=/opt/local/fast;` to `/opt/pgi/linux86-64/13.10/bin/siterc`
Make available to all compilations the include path `-I/opt/acml/include`	`set SITEINC=/opt/acml/include;` to `/opt/pgi/linux86/13.10/bin/siterc` and `/opt/pgi/linux86-64/13.10/bin/siterc`

With linux86-64 compilations, change –Mmpi to link in `/opt/mympi/64/libmpix.a`	`set MPILIBDIR=/opt/mympi/64;`
	`set MPILIBNAME=mpix;`
	to `/opt/pgi/linux86-64/13.10/bin/siterc;`

Have linux86-64 compilations always add `–DIS64BIT –DAMD`	`set SITEDEF=IS64BIT AMD;` to `/opt/pgi/linux86-64/13.10/bin/siterc`
Build an F90 or F95 executable for linux86-64 or linux86 that resolves PGI shared objects in the relative directory `./REDIST`	`set RPATH=./REDIST ;` to `~/.mypgfortranrc` **Note.** This only affects the behavior of PGFORTRAN for the given user.

Common Development Tasks

Now that you have a brief introduction to the compiler, let's look at some common development tasks that you might wish to perform.

- When you compile code you can specify a number of options on the command line that define specific characteristics related to how the program is compiled and linked, typically enhancing or overriding the default behavior of the compiler. For a list of the most common command line options and information on all the command line options, refer to Chapter 2, "*Using Command Line Options*".
- Code optimization and parallelization allows the compiler to organize your code for efficient execution. While possibly increasing compilation time and making the code more difficult to debug, these techniques typically produce code that runs significantly faster than code that does not use them. For more information on optimization and parallelization, refer to Chapter 3, "*Optimizing & Parallelizing*".
- Function inlining, a special type of optimization, replaces a call to a function or a subroutine with the body of the function or subroutine. This process can speed up execution by eliminating parameter passing and

the function or subroutine call and return overhead. In addition, function inlining allows the compiler to optimize the function with the rest of the code. However, function inlining may also result in much larger code size with no increase in execution speed. For more information on function inlining, refer to Chapter 4, "*Using Function Inlining*".

- Directives and pragmas allow users to place hints in the source code to help the compiler generate better assembly code. You typically use directives and pragmas to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. You place them in your source code where you want them to take effect. A directive or pragma typically stays in effect from the point where included until the end of the compilation unit or until another directive or pragma changes its status. For more information on directives and pragmas, refer to Chapter 5, "*Using OpenMP*" and Chapter 9, "*Using Directives and Pragmas*".
- A library is a collection of functions or subprograms used to develop software. Libraries contain "helper" code and data, which provide services to independent programs, allowing code and data to be shared and changed in a modular fashion. The functions and programs in a library are grouped for ease of use and linking. When creating your programs, it is often useful to incorporate standard libraries or proprietary ones. For more information on this topic, refer to Chapter 10, "*Creating and Using Libraries*".
- Environment variables define a set of dynamic values that can affect the way running processes behave on a computer. It is often useful to use these variables to set and pass information that alters the default behavior of the PGI compilers and the executables which they generate. For more information on these variables, refer to Chapter 11, " *Using Environment Variables*".
- Deployment, though possibly an infrequent task, can present some unique issues related to concerns of porting the code to other systems. Deployment, in this context, involves distribution of a specific file or set of files that are already compiled and configured. The distribution must occur in such a way that the application executes accurately on another system which may not be configured exactly the same as the system on which the code was created. For more information on what you might need to know to successfully deploy your code, refer to Chapter 12, "*Distributing Files - Deployment*".
- An intrinsic is a function available in a given language whose implementation is handled specially by the compiler. Intrinsics make using processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages details that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data. For C/C++ programs, PGI provides support for MMX SSE, SSE2, SSE3, SSSE3, SSE4A, ABM, and AVX intrinsics. For more information on these intrinsics, refer to the "C/C++ MMX/SSE Inline Intrinsics" chapter of the PGI Compiler Reference Manual.

Chapter 2. Using Command Line Options

A command line option allows you to control specific behavior when a program is compiled and linked. This chapter describes the syntax for properly using command-line options and provides a brief overview of a few of the more common options.

Note

For a complete list of command-line options, their descriptions and use, refer to the "Command-Line Options Reference" chapter in the PGI Compiler Reference Manual.

Command Line Option Overview

Before looking at all the command-line options, first become familiar with the syntax for these options. There are a large number of options available to you, yet most users only use a few of them. So, start simple and progress into using the more advanced options.

By default, the PGI compilers generate code that is optimized for the type of processor on which compilation is performed, the compilation host. Before adding options to your command-line, review the sections "Help with Command-line Options," on page 18 and "Frequently-used Options," on page 21.

Command-line Options Syntax

On a command-line, options need to be preceded by a hyphen (-). If the compiler does not recognize an option, you get an unknown switch error. The error can be downgraded to a warning by adding the `-noswitcherror` option.

This document uses the following notation when describing options:

[item]
: Square brackets indicate that the enclosed item is optional.

{item | item}
: Braces indicate that you must select one and only one of the enclosed items. A vertical bar (|) separates the choices.

...

Horizontal ellipses indicate that zero or more instances of the preceding item are valid.

NOTE

Some options do not allow a space between the option and its argument or within an argument. When applicable, the syntax section of the option description in the "Command-Line Options Reference" chapter in the PGI Compiler Reference Manual contains this information.

Command-line Suboptions

Some options accept several suboptions. You can specify these suboptions either by using the full option statement multiple times or by using a comma-separated list for the suboptions.

The following two command lines are equivalent:

```
pgfortran -Mvect=simd -Mvect=noaltcode
```

```
pgfortran -Mvect=simd,noaltcode
```

Command-line Conflicting Options

Some options have an opposite or negated counterpart. For example, both `-Mvect` and `-Mnovect` are available. `-Mvect` enables vectorization and `-Mnovect` disables it. If you used both of these commands on a command line, they would conflict.

Note

Rule: When you use conflicting options on a command line, the last encountered option takes precedence over any previous one.

This rule is important for a number of reasons.

- Some options, such as `-fast`, include other options. Therefore, it is possible for you to be unaware that you have conflicting options.
- You can use this rule to create makefiles that apply specific flags to a set of files, as shown in Example 2.1.

Example 2.1. Makefiles with Options

In this makefile fragment, CCFLAGS uses vectorization. CCNOVECTFLAGS uses the flags defined for CCFLAGS but disables vectorization.

```
CCFLAGS=c -Mvect=simd
CCNOVECTFLAGS=$(CCFLAGS) -Mnovect
```

Help with Command-line Options

If you are just getting started with the PGI compilers and tools, it is helpful to know which options are available, when to use them, and which options most users find effective.

Using –help

The `–help` option is useful because it provides information about all options supported by a given compiler. You can use `–help` in one of three ways:

- Use `–help` with no parameters to obtain a list of all the available options with a brief one-line description of each.
- Add a parameter to `–help` to restrict the output to information about a specific option. The syntax for this usage is this:

  ```
  –help <command line option>
  ```

  Suppose you use the following command to restrict the output to information about the `-fast` option:

  ```
  $ pgfortran -help -fast
  ```

  The output you see is similar to this:

  ```
  -fast Common optimizations; includes -O2 -Munroll=c:1 -Mnoframe -Mlre
  ```

  In the following example, we add the `–help` parameter to restrict the output to information about the help command. The usage information for `–help` shows how groups of options can be listed or examined according to function.

  ```
  $ pgfortran -help -help
  -help[=groups|asm|debug|language|linker|opt|other|
   overall|phase|prepro|suffix|switch|target|variable]
   Show compiler switches
  ```

- Add a parameter to `–help` to restrict the output to a specific set of options or to a building process. The syntax for this usage is this:

  ```
  -help=<subgroup>
  ```

  By using the command `pgfortran -help -help`, as previously shown, we can see output that shows the available subgroups. You can use the following command to restrict the output on the `–help` command to information about only the options related to only one group, such as debug information generation.

  ```
  $ pgfortran -help=debug
  ```

  The output you see is similar to this:

  ```
  Debugging switches:
  -M[no]bounds Generate code to check array bounds
  -Mchkfpstk Check consistency of floating point stack at subprogram calls
   (32-bit only)
  -Mchkstk Check for sufficient stack space upon subprogram entry
  -Mcoff Generate COFF format object
  -Mdwarf1 Generate DWARF1 debug information with -g
  -Mdwarf2 Generate DWARF2 debug information with -g
  -Mdwarf3 Generate DWARF3 debug information with -g
  -Melf Generate ELF format object
  -g Generate information for debugger
  -gopt Generate information for debugger without disabling
   optimizations
  ```

  For a complete description of subgroups, refer to the "–help" description in the Command Line Options Reference chapter of the PGI Compiler Reference Manual.

Getting Started with Performance

One of the top priorities of most users is performance and optimization. This section provides a quick overview of a few of the command-line options that are useful in improving performance.

Using –fast and –fastsse Options

PGI compilers implement a wide range of options that allow users a fine degree of control on each optimization phase. When it comes to optimization of code, the quickest way to start is to use the options `–fast` and `–fastsse`. These options create a generally optimal set of flags for x86 targets They incorporate optimization options to enable use of vector streaming SIMD (SSE) instructions for 64-bit targets. They enable vectorization with SSE instructions, cache alignment, and SSE arithmetic to flush to zero mode.

Note

The contents of the `–fast` and `–fastsse` options are host-dependent. Further, you should use these options on both compile and link command lines.

- `–fast` and `–fastsse` typically include these options:

–O2	Specifies a code optimization level of 2.
–Munroll=c:1	Unrolls loops, executing multiple instances of the loop during each iteration.
–Mnoframe	Indicates to not generate code to set up a stack frame.
–Mlre	Indicates loop-carried redundancy elimination.
–Mpre	Indicates partial redundancy elimination.

- These additional options are also typically available when using `–fast` for 64-bit targets or `–fastsse` for both 32- and 64-bit targets:

–Mvect=simd	Generates SSE instructions.
–Mcache_align	Aligns long objects on cache-line boundaries.
–Mflushz	Sets SSE to flush-to-zero mode.

Note

For best performance on processors that support SSE instructions, use the PGFORTRAN compiler, even for FORTRAN 77 code, and the `–fast` option.

To see the specific behavior of `–fast` for your target, use the following command:

```
$ pgfortran -help -fast
```

Other Performance-related Options

While `-fast` and `-fastsse` are options designed to be the quickest route to best performance, they are limited to routine boundaries. Depending on the nature and writing style of the source code, the compiler often can perform further optimization by knowing the global context of usage of a given routine. For instance, determining the possible value range of actual parameters of a routine could enable a loop to be vectorized; similarly, determining static occurrence of calls helps to decide which routine is beneficial to inline.

These types of global optimizations are under control of Interprocedural Analysis (IPA) in PGI compilers. Option `-Mipa` enables Interprocedural Analysis. `-Mpi=fast` is the recommended option to get best performances for global optimization. You can also add the suboption `inline` to enable automatic global inlining across files. You might consider using `-Mipa=fast,inline`. This option for interprocedural analysis and global optimization can improve performance.

You may also obtain further performance improvements by experimenting with the `-M<pgflag>` options described in the section "–M Options by Category" section of the PGI Compiler Reference Manual. These options include, but are not limited to, `-Mconcur`, `-Mvect`, `-Munroll`, `-Minline`, and `-Mpfi/-Mpfo`. However, performance improvements using these options are typically application- and system-dependent. It is important to time your application carefully when using these options to ensure no performance degradations occur.

For more information on optimization, refer to Chapter 3, "*Optimizing & Parallelizing*". For specific information about these options, refer to the "Optimization Controls" section in the PGI Compiler Reference Manual.

Targeting Multiple Systems - Using the -tp Option

The `-tp` option allows you to set the target architecture. By default, the PGI compiler uses all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

Processor-specific optimizations can be specified or limited explicitly by using the `-tp` option. Thus, it is possible to create executables that are usable on previous generation systems. Using a `-tp` flag option of k8 or p7 produces an executable that runs on most x86 hardware in use today.

For more information about the `-tp` option, refer to "–tp <target> [,target...]" description in the "Command-Line Options Reference" chapter in the PGI Compiler Reference Manual.

Frequently-used Options

In addition to overall performance, there are a number of other options that many users find useful when getting started. The following table provides a brief summary of these options.

For more information on these options, refer to the complete description of each option available in the "Command-Line Options Reference" chapter in the PGI Compiler Reference Manual. Also, there are a number of suboptions available with each of the `-M` options listed. For more information on those options, refer to the "–M Options by Category" section within that chapter.

Table 2.1. Commonly Used Command Line Options

Option	Description
–fast –fastsse	These options create a generally optimal set of flags for targets that support SIMD capability. They incorporate optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization with SEE instructions, cache aligned and flushz.
–g	Instructs the compiler to include symbolic debugging information in the object module.
–gopt	Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.
–help	Provides information about available options.
–mcmodel=medium	Enables medium=model core generation for 64-bit targets; useful when the data space of the program exceeds 4GB.
–Mconcur	Instructs the compiler to enable auto-concurrentization of loops. If specified, the compiler uses multiple processors to execute loops that it determines to be parallelizable; thus, loop iterations are split to execute optimally in a multithreaded execution context.
–Minfo	Instructs the compiler to produce information on standard error.
–Minline	Enables function inlining.
–Mipa=fast,inline	Enables interprocedural analysis and optimization. Also enables automatic procedure inlining.
–Mpfi or –Mpfo	Enable profile feedback driven optimizations.
–Mkeepasm	Keeps the generated assembly files.
–Munroll	Invokes the loop unroller to unroll loops, executing multiple instances of the loop during each iteration. This also sets the optimization level to 2 if the level is set to less than 2, or if no –O or –g options are supplied.
–M[no]vect	Enables/Disables the code vectorizer.
--[no_]exceptions	Removes exception handling from user code. For C++, declares that the functions in this file generate no C++ exceptions, allowing more optimal code generation.
–o	Names the output file.
–O<level>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
–tp <target> [,target...]	Specify the target processor(s); for the 64-bit compilers, more than one target is allowed, and enables generation of PGI Unified Binary executables.
–W1, <option>	Compiler driver passes the specified options to the linker.

Chapter 3. Optimizing & Parallelizing

Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

Invoking one of the PGI compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

Note

PGI provides a profiler, PGPROF, that provides a way to visualize the performance of the components of your program. Using tables and graphs, PGPROF associates execution time and resource utilization data with the source code and instructions of your program, allowing you to see where execution time is spent. Through resource utilization data and compiler analysis information, PGPROF helps you to understand why certain parts of your program have high execution times.

The compilers optimize code according to the specified optimization level. You can use a number of options to specify the optimization levels, including `-O`, `-Mvect`, `-Mipa`, and `-Mconcur`. In addition, you can use several of the –M<pgflag> switches to control specific types of optimization and parallelization.

This chapter describes the optimization options displayed in the following list.

`-fast`	`-Minline`	`-Mpfi`	`-Mvect`
`-Mconcur`	`-Mipa=fast`	`-Mpfo`	`-O`
`-Minfo`	`-Mneginfo`	`-Munroll`	`-Msafeptr`

This chapter also describes how to choose optimization options to use with the PGI compilers. This overview will help if you are just getting started with one of the PGI compilers, or wish to experiment with individual optimizations. Complete specifications of each of these options is available in the "Command-Line Options Reference" chapter of the PGI Compiler Reference Manual.

Overview of Optimization

In general, optimization involves using transformations and replacements that generate more efficient code. This is done by the compiler and involves replacements that are independent of the particular target processor's architecture as well as replacements that take advantage of the x86 or x64 architecture, instruction set and registers. For the discussion in this and the following chapters, optimization is divided into the following categories:

Local Optimization

This optimization is performed on a block-by-block basis within a program's basic blocks. A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end. The PGI compilers perform many types of local optimization including: algebraic identity removal, constant folding, common sub-expression elimination, redundant load and store elimination, scheduling, strength reduction, and peephole optimizations.

Global Optimization

This optimization is performed on a program unit over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by IFs and GOTOs, are detected and optimized. Global optimization includes: constant propagation, copy propagation, dead store elimination, global register allocation, invariant code motion, and induction variable elimination.

Loop Optimization: Unrolling, Vectorization, and Parallelization

The performance of certain classes of loops may be improved through vectorization or unrolling options. Vectorization transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

Interprocedural Analysis (IPA) and Optimization

Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable. A wide range of optimizations are enabled or improved by using IPA, including but not limited to data alignment optimizations, argument removal, constant propagation, pointer disambiguation, pure function detection, F90/F95 array shape propagation, data placement, vestigial function removal, automatic function inlining, inlining of functions from pre-compiled libraries, and interprocedural optimization of functions from pre-compiled libraries.

Function Inlining

This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead. Function inlining may also create opportunities for other types of optimization. Function inlining is not always beneficial. When used improperly it may increase code size and generate less efficient code.

Profile-Feedback Optimization (PFO)

Profile-feedback optimization (PFO) makes use of information from a trace file produced by specially instrumented executables which capture and save information on branch frequency, function and subroutine call frequency, semi-invariant values, loop index ranges, and other input data dependent information that can only be collected dynamically during execution of a program. By definition, use of profile-feedback optimization is a two-phase process: compilation and execution of a specially-instrumented executable, followed by a subsequent compilation which reads a trace file generated during the first phase and uses the information in that trace file to guide compiler optimizations.

Getting Started with Optimizations

Your first concern should be getting your program to execute and produce correct results. To get your program running, start by compiling and linking without optimization. Use the optimization level `–O0` or select `–g` to perform minimal optimization. At this level, you will be able to debug your program easily and isolate any coding errors exposed during porting to x86 or x64 platforms.

If you want to get started quickly with optimization, a good set of options to use with any of the PGI compilers is `–fast –Mipa=fast`. For example:

```
$ pgfortran -fast -Mipa=fast prog.f
```

For all of the PGI Fortran, C, and C++ compilers, the `–fast –Mipa=fast` options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.

- The `–fast` option is an aggregate option that includes a number of individual PGI compiler options; which PGI compiler options are included depends on the target for which compilation is performed.
- The `–Mipa=fast` option invokes interprocedural analysis including several IPA suboptions.
- For C++ programs, add `-Minline=levels:10 --no_exceptions` as shown here:

```
$ pgcpp -fast -Mipa=fast -Minline=levels:10 --no_exceptions prog.cc
```

Note

A C++ program compiled with `--no_exceptions` fails if the program uses exception handling.

By experimenting with individual compiler options on a file-by-file basis, further significant performance gains can sometimes be realized. However, depending on the coding style, individual optimizations can sometimes cause slowdowns, and must be used carefully to ensure performance improvements. In addition to `-fast`, the optimization flags most likely to further improve performance are `-O3`, `-Mpfi`, `-Mpfo`, `-Minline`; and on targets with multiple processors, you can use `-Mconcur`.

In addition, the `–Msafeptr` option can significantly improve performance of C/C++ programs in which there is known to be no pointer aliasing. For obvious reasons this command-line option must be used carefully.

Three other extremely useful options are `-help`, `-Minfo`, and `-dryrun`.

–help

As described in "Help with Command-line Options," on page 18, you can see a specification of any command-line option by invoking any of the PGI compilers with `-help` in combination with the option in question, without specifying any input files.

For example, you might want information on `-O`:

```
$ pgfortran -help -O
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi/linux86-64/7.0/bin/.pgfortranrc
-O[<n>] Set optimization level, -O0 to -O4, default -O2
```

Or you can see the full functionality of `-help` itself, which can return information on either an individual option or groups of options:

```
$ pgfortran -help -help
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi_rel/linux86-64/7.0/bin/.pgfortranrc
-help[=groups|asm|debug|language|linker|opt|other|overall|
  phase|prepro|suffix|switch|target|variable]
```

–Minfo

You can use the `-Minfo` option to display compile-time optimization listings. When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, SSE instructions, vectorization, parallelization, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

For more information on `–Minfo`, refer to "Optimization Controls" in the PGI Compiler Reference Manual.

–Mneginfo

You can use the `-Mneginfo` option to display informational messages listing why certain optimizations are inhibited.

For more information on `–Mneginfo`, refer to "Optimization Controls" in the PGI Compiler Reference Manual.

–dryrun

The `–dryrun` option can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs. When you specify the `–dryrun` option, these steps will be printed to stderr but are not actually performed. For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

The remainder of this chapter describes the `–0` options, the loop unroller option `–Munroll`, the vectorizer option `–Mvect`, the auto-parallelization option `–Mconcur`, the interprocedural analysis optimization `–Mipa`, and the profile-feedback instrumentation (`–Mpfi`) and optimization (`–Mpfo`) options. You should be able to get very near optimal compiled performance using some combination of these switches.

Common Compiler Feedback Format (CCFF)

Using the Common Compiler Feedback Format (CCFF), PGI compilers save information about how your program was optimized, or why a particular optimization was not made, in the executable file. To append this information to the object file, use the compiler option `-Minfo=ccff`.

If you choose to use PGPROF to aid with your optimization, PGPROF can extract this information and associate it with source code and other performance data, allowing you to view all of this information simultaneously in one of the available profiler panels.

Local and Global Optimization using -O

Using the PGI compiler commands with the –Olevel option (the capital O is for Optimize), you can specify any of the following optimization levels:

–O0
: Level zero specifies no optimization. A basic block is generated for each language statement. At this level, the compiler generates a basic block for each statement.

  Performance will almost always be slowest using this optimization level. This level is useful for the initial execution of a program. It is also useful for debugging, since there is a direct correlation between the program text and the code generated.

–O1
: Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

  Local optimization is a good choice when the code is very irregular, such as code that contains many short statements containing IF statements and does not contain loops (DO or DO WHILE statements). Although this case rarely occurs, for certain types of code, this optimization level may perform better than level-two (`-O2`).

–O
: When no level is specified, level two global optimizations are performed, including traditional scalar optimizations, induction recognition, and loop invariant motion. No SIMD vectorization is enabled.

–O2
: Level two specifies global optimization. This level performs all level-one local optimization as well as level-two global optimization described in –O. In addition, more advanced optimizations such as SIMD code generation, cache alignment, and partial redundancy elimination are enabled.

–O3
: Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

–O4
: Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

The PGI compilers perform many different types of local optimizations, including but not limited to:

- Algebraic identity removal
- Constant folding
- Common subexpression elimination
- Local register optimization
- Peephole optimizations
- Redundant load and store elimination
- Strength reductions

Level-two optimization (`-O2` or `-O`) specifies global optimization. The `-fast` option generally will specify global optimization; however, the `-fast` switch varies from release to release, depending on a reasonable selection of switches for any one particular release. The –O or –O2 level performs all level-one local optimizations as well as global optimizations. Control flow analysis is applied and global registers are allocated for all functions and subroutines. Loop regions are given special consideration. This optimization level is a good choice when the program contains loops, the loops are short, and the structure of the code is regular.

The PGI compilers perform many different types of global optimizations, including but not limited to:

- Branch to branch elimination
- Constant propagation
- Copy propagation
- Dead store elimination
- Global register allocation
- Invariant code motion
- Induction variable elimination

You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:

```
$ pgfortran -O2 prog.f
```

The default optimization level changes depending on which options you select on the command line. For example, when you select the `-g` debugging option, the default optimization level is set to level-zero (`-O0`). However, if you need to debug optimized code, you can use the `-gopt` option to generate debug information without perturbing optimization. For a description of the default levels, refer to "Default Optimization Levels," on page 43.

As noted previously, the `-fast` option includes `-O2` on all x86 and x64 targets. If you want to override the default for`-fast` with `-O3` while maintaining all other elements of `-fast`, simply compile as follows:

```
$ pgfortran -fast -O3 prog.f
```

Loop Unrolling using –Munroll

This optimization unrolls loops, executing multiple instances of the loop during each iteration. This reduces branch overhead, and can improve execution speed by creating better opportunities for instruction scheduling. A loop with a constant count may be completely unrolled or partially unrolled. A loop with a non-constant count may also be unrolled. A candidate loop must be an innermost loop containing one to four blocks of code.

The following example shows the use of the `-Munroll` option:

```
$ pgfortran -Munroll prog.f
```

The `-Munroll` option is included as part of `-fast` on all x86 and x64 targets. The loop unroller expands the contents of a loop and reduces the number of times a loop is executed. Branching overhead is reduced when

a loop is unrolled two or more times, since each iteration of the unrolled loop corresponds to two or more iterations of the original loop; the number of branch instructions executed is proportionately reduced. When a loop is unrolled completely, the loop's branch overhead is eliminated altogether.

Loop unrolling may be beneficial for the instruction scheduler. When a loop is completely unrolled or unrolled two or more times, opportunities for improved scheduling may be presented. The code generator can take advantage of more possibilities for instruction grouping or filling instruction delays found within the loop.

Example 3.1, "Dot Product Code" and Example 3.2, "Unrolled Dot Product Code" show the effect of code unrolling on a segment that computes a dot product.

Note

This example is only meant to represent how the compiler can transform the loop; it is not meant to imply that the programmer needs to manually change code. In fact, manually unrolling your code can sometimes inhibit the compiler's analysis and optimization.

Example 3.1. Dot Product Code

```
REAL*4 A(100), B(100), Z
INTEGER I
DO I=1, 100
  Z = Z + A(i) * B(i)
END DO
END
```

Example 3.2. Unrolled Dot Product Code

```
REAL*4 A(100), B(100), Z
 INTEGER I
 DO I=1, 100, 2
   Z = Z + A(i) * B(i)
   Z = Z + A(i+1) * B(i+1)
 END DO
END
```

Using the `-Minfo` option, the compiler informs you when a loop is being unrolled. For example, a message similar to the following, indicating the line number, and the number of times the code is unrolled, displays when a loop is unrolled:

```
dot:
 5, Loop unrolled 5 times
```

Using the c:<m> and n:<m> sub-options to `-Munroll`, or using `-Mnounroll`, you can control whether and how loops are unrolled on a file-by-file basis. Using directives or pragmas as specified in Chapter 9, "*Using Directives and Pragmas*", you can precisely control whether and how a given loop is unrolled. For a detailed description of the `-Munroll` option, refer to Chapter 2, "*Using Command Line Options*".

Vectorization using –Mvect

The `-Mvect` option is included as part of `-fast` on all x86 and x64 targets. If your program contains computationally-intensive loops, the `-Mvect` option may be helpful. If in addition you specify `-Minfo`, and your code contains loops that can be vectorized, the compiler reports relevant information on the optimizations applied.

When a PGI compiler command is invoked with the `-Mvect` option, the vectorizer scans code searching for loops that are candidates for high-level transformations such as loop distribution, loop interchange, cache tiling, and idiom recognition (replacement of a recognizable code sequence, such as a reduction loop, with optimized code sequences or function calls). When the vectorizer finds vectorization opportunities, it internally rearranges or replaces sections of loops (the vectorizer changes the code generated; your source code's loops

are not altered). In addition to performing these loop transformations, the vectorizer produces extensive data dependence information for use by other phases of compilation and detects opportunities to use vector or packed Streaming SIMD Extensions (SSE) instructions on processors where these are supported.

The `-Mvect` option can speed up code which contains well-behaved countable loops which operate on large REAL, REAL(4), REAL(8), INTEGER, INTEGER(4), COMPLEX(4) or COMPLEX(8) arrays in Fortran and their C/C++ counterparts. However, it is possible that some codes will show a decrease in performance when compiled with the `-Mvect` option due to the generation of conditionally executed code segments, inability to determine data alignment, and other code generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled with this option enabled.

Vectorization Sub-options

The vectorizer performs high-level loop transformations on countable loops. A loop is countable if the number of iterations is set only before loop execution and cannot be modified during loop execution. Some of the vectorizer transformations can be controlled by arguments to the `-Mvect` command line option. The following sections describe the arguments that affect the operation of the vectorizer. In addition, some of these vectorizer operations can be controlled from within code using directives and pragmas. For details on the use of directives and pragmas, refer to Chapter 9, "*Using Directives and Pragmas*".

The vectorizer performs the following operations:

- Loop interchange
- Loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SSE instructions on processors where these are supported
- Generation of prefetch instructions on processors where these are supported
- Loop iteration peeling to maximize vector alignment
- Alternate code generation

By default, `-Mvect` without any sub-options is equivalent to:

```
-Mvect=assoc,cachesize=c
```

where c is the actual cache size of the machine.

This enables the options for nested loop transformation and various other vectorizer options. These defaults may vary depending on the target system.

Assoc Option

The option `-Mvect=assoc` instructs the vectorizer to perform associativity conversions that can change the results of a computation due to a round-off error (`-Mvect=noassoc` disables this option). For example, a typical optimization is to change one arithmetic operation to another arithmetic operation that

is mathematically correct, but can be computationally different and generate faster code. This option is provided to enable or disable this transformation, since a round-off error for such associativity conversions may produce unacceptable results.

Cachesize Option

The option `-Mvect=cachesize:n` instructs the vectorizer to tile nested loop operations assuming a data cache size of n bytes. By default, the vectorizer attempts to tile nested loop operations, such as matrix multiply, using multi-dimensional strip-mining techniques to maximize re-use of items in the data cache.

SIMD Option

The option `-Mvect=simd` instructs the vectorizer to automatically generate packed SSE (Streaming SIMD Extensions), and prefetch instructions when vectorizable loops are encountered. SIMD instructions, first introduced on Pentium III and AthlonXP processors, operate on single-precision floating-point data.

Prefetch Option

The option `-Mvect=prefetch` instructs the vectorizer to automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SSE instructions are not generated.

In addition to these sub-options to `-Mvect`, several other sub-options are supported. For a detailed description of all available sub-options, refer to the description of `-M[no]vect` in the "Command-Line Options Reference" section of the PGI Compiler Reference Manual.

Vectorization Example Using SIMD Instructions

One of the most important vectorization options is `-Mvect=simd`. When you use this option, the compiler automatically generates SSE instructions, where possible, when targeting processors on which these instructions are supported. This process can improve performance by several factors compared with the equivalent scalar code. All of the PGI Fortran, C and C++ compilers support this capability. The PGI Release Notes show which x86 and x64 processors support these instructions.

In the program in Example 3.3, "Vector operation using SIMD instructions", the vectorizer recognizes the vector operation in subroutine 'loop' when either the compiler switch `-Mvect=simd` or `-fast` is used. This example shows the compilation, informational messages, and runtime results using the SSE instructions on an AMD Opteron processor-based system, along with issues that affect SSE performance.

Loops vectorized using SSE instructions operate much more efficiently when processing vectors that are aligned to a cache-line boundary. You can cause unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the `-Mcache_align` switch. An unconstrained data object is a data object that is not a common block member and not a member of an aggregate data structure.

Note

For stack-based local variables to be properly aligned, the main program or function must be compiled with `-Mcache_align`.

The `-Mcache_align` switch has no effect on the alignment of Fortran allocatable or automatic arrays. If you have arrays that are constrained, such as vectors that are members of Fortran common blocks, you must

specifically pad your data structures to ensure proper cache alignment. You can use `–Mcache_align` for only the beginning address of each common block to be cache-aligned.

The following examples show the results of compiling the sample code in Example 3.3 both with and without the option `–Mvect=simd`.

Example 3.3. Vector operation using SIMD instructions

```
program vector_op
 parameter (N = 9999)
 real*4 x(N), y(N), z(N), W(N)
 do i = 1, n
   y(i) = i
   z(i) = 2*i
   w(i) = 4*i
 enddo
 do j = 1, 200000
   call loop(x,y,z,w,1.0e0,N)
 enddo
 print *, x(1),x(771),x(3618),x(6498),x(9999)
end
```

```
subroutine loop(a,b,c,d,s,n)
 integer i, n
 real*4 a(n), b(n), c(n), d(n),s
 do i = 1, n
   a(i) = b(i) + c(i) - s * d(i)
 enddo
end
```

Assume the preceding program is compiled as follows, where `-Mvect=nosse` disables SSE vectorization:

```
% pgfortran -fast -Mvect=nosse -Minfo vadd.f
 vector_op:
 4, Loop unrolled 4 times
 loop:
 18, Loop unrolled 4 times
```

The following output shows a sample result if the generated executable is run and timed on a standalone AMD Opteron 2.2 Ghz system:

```
 % /bin/time vadd
 -1.000000 -771.000 -3618.000 -6498.00 -9999.00
 5.39user 0.00system 0:05.40elapsed 99%CP
```

Now, recompile with SSE vectorization enabled, and you see results similar to these:

```
 % pgfortran -fast -Minfo vadd.f -o vadd
 vector_op:
 4, Unrolled inner loop 8 times
 Loop unrolled 7 times (completely unrolled)
 loop:
 18, Generated 4 alternate loops for the inner loop
 Generated vector sse code for inner loop
 Generated 3 prefetch instructions for this loop
```

Notice the informational message for the loop at line 18.

- The first two lines of the message indicate that the loop was vectorized, SSE instructions were generated, and four alternate versions of the loop were also generated. The loop count and alignments of the arrays determine which of these versions is executed.
- The last line of the informational message indicates that prefetch instructions have been generated for three loads to minimize latency of data transfers from main memory.

Executing again, you should see results similar to the following:

```
% /bin/time vadd
-1.000000 -771.000 -3618.00 -6498.00
-9999.0
3.59user 0.00system 0:03.59elapsed 100%CPU
```

The result is a 50% speed-up over the equivalent scalar, that is, the non-SSE, version of the program.

Speed-up realized by a given loop or program can vary widely based on a number of factors:

- When the vectors of data are resident in the data cache, performance improvement using vector SSE or SSE2 instructions is most effective.
- If data is aligned properly, performance will be better in general than when using vector SSE operations on unaligned data.
- If the compiler can guarantee that data is aligned properly, even more efficient sequences of SSE instructions can be generated.
- The efficiency of loops that operate on single-precision data can be higher. SSE2 vector instructions can operate on four single-precision elements concurrently, but only two double-precision elements.

> **Note**
>
> Compiling with `–Mvect-simd` can result in numerical differences from the executables generated with less optimization. Certain vectorizable operations, for example dot products, are sensitive to order of operations and the associative transformations necessary to enable vectorization (or parallelization).

Auto-Parallelization using -Mconcur

With the `-Mconcur` option the compiler scans code searching for loops that are candidates for auto-parallelization. `-Mconcur` must be used at both compile-time and link-time. When the parallelizer finds opportunities for auto-parallelization, it parallelizes loops and you are informed of the line or loop being parallelized if the `-Minfo` option is present on the compile line. Refer to "Optimization Controls" in the PGI Compiler Reference Guide for a complete specification of `-Mconcur`.

A loop is considered parallelizable if doesn't contain any cross-iteration data dependencies. Cross-iteration dependencies from reductions and expandable scalars are excluded from consideration, enabling more loops to be parallelizable. In general, loops with calls are not parallelized due to unknown side effects. Also, loops with low trip counts are not parallelized since the overhead in setting up and starting a parallel loop will likely outweigh the potential benefits. In addition, the default is to not parallelize innermost loops, since these often by definition are vectorizable using SSE instructions and it is seldom profitable to both vectorize and parallelize

the same loop, especially on multi-core processors. Compiler switches and directives are available to let you override most of these restrictions on auto-parallelization.

Auto-parallelization Sub-options

The parallelizer performs various operations that can be controlled by arguments to the `-Mconcur` command line option. The following sections describe these arguments that affect the operation of the vectorizer. In addition, these vectorizer operations can be controlled from within code using directives and pragmas.

For details on the use of directives and pragmas, refer to Chapter 9, "*Using Directives and Pragmas*".

By default, `-Mconcur` without any sub-options is equivalent to:

```
-Mconcur=dist:block
```

This enables parallelization of loops with blocked iteration allocation across the available threads of execution. These defaults may vary depending on the target system.

Altcode Option

The option `-Mconcur=altcode` instructs the parallelizer to generate alternate serial code for parallelized loops. If altcode is specified without arguments, the parallelizer determines an appropriate cutoff length and generates serial code to be executed whenever the loop count is less than or equal to that length. If `altcode:n` is specified, the serial altcode is executed whenever the loop count is less than or equal to `n`. If `noaltcode` is specified, no alternate serial code is generated.

Dist Option

The option `-Mconcur=dist:{block|cyclic}` option specifies whether to assign loop iterations to the available threads in blocks or in a cyclic (round-robin) fashion. Block distribution is the default. If cyclic is specified, iterations are allocated to processors cyclically. That is, processor 0 performs iterations 0, 3, 6, etc.; processor 1 performs iterations 1, 4, 7, etc.; and processor 2 performs iterations 2, 5, 8, etc.

Cncall Option

The option `-Mconcur=cncall` specifies that it is safe to parallelize loops that contain subroutine or function calls. By default, such loops are excluded from consideration for auto-parallelization. Also, no minimum loop count threshold must be satisfied before parallelization will occur, and last values of scalars are assumed to be safe.

The environment variable `NCPUS` is checked at runtime for a parallel program. If `NCPUS` is set to 1, a parallel program runs serially, but will use the parallel routines generated during compilation. If `NCPUS` is set to a value greater than 1, the specified number of processors will be used to execute the program. Setting `NCPUS` to a value exceeding the number of physical processors can produce inefficient execution. Executing a program on multiple processors in an environment where some of the processors are being time-shared with another executing job can also result in inefficient execution.

As with the vectorizer, the `-Mconcur` option can speed up code if it contains well-behaved countable loops and/or computationally intensive nested loops that operate on arrays. However, it is possible that some codes will show a decrease in performance on multi-processor systems when compiled with `-Mconcur` due to parallelization overheads, memory bandwidth limitations in the target system, false-sharing of cache lines, or

other architectural or code-generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled using this option.

If the compiler is not able to successfully auto-parallelize your application, you should refer to Chapter 5, "*Using OpenMP*". It is possible that insertion of explicit parallelization directives or pragmas, and use of the compiler option `-mp` might enable the application to run in parallel.

Loops That Fail to Parallelize

In spite of the sophisticated analysis and transformations performed by the compiler, programmers may notice loops that are seemingly parallel, but are not parallelized. In this subsection, we look at some examples of common situations where parallelization does not occur.

Innermost Loops

As noted earlier in this chapter, the PGI compilers will not parallelize innermost loops by default, because it is usually not profitable. However, you can override this default using the `-Mconcur=innermost` command-line option.

Timing Loops

Often, loops occur in programs that are similar to timing loops. The outer loop in the following example is one such loop.

```
do j = 1, 2
 do i = 1, n
 a(i) = b(i) + c(i)
1 enddo
enddo
```

The outer loop in the preceding example is not parallelized because the compiler detects a cross-iteration dependence in the assignment to `a(i)`. Suppose the outer loop were parallelized. Then both processors would simultaneously attempt to make assignments into `a(1:n)`. Now in general the values computed by each processor for `a(1:n)` will differ, so that simultaneous assignment into `a(1:n)` will produce values different from sequential execution of the loops.

In this example, values computed for `a(1:n)` don't depend on `j`, so that simultaneous assignment by both processors will not yield incorrect results. However, it is beyond the scope of the compilers' dependence analysis to determine that values computed in one iteration of a loop don't differ from values computed in another iteration. So the worst case is assumed, and different iterations of the outer loop are assumed to compute different values for `a(1:n)`. Is this assumption too pessimistic? If `j` doesn't occur anywhere within a loop, the loop exists only to cause some delay, most probably to improve timing resolution. It is not usually valid to parallelize timing loops; to do so would distort the timing information for the inner loops.

Scalars

Quite often, scalars will inhibit parallelization of non-innermost loops. There are two separate cases that present problems. In the first case, scalars appear to be expandable, but appear in non-innermost loops, as in the following example.

```
do j = 1, n
 x = b(j)
```

```
 do i = 1, n
 a(i,j) = x + c(i,j)
 enddo
enddo
```

There are a number of technical problems to be resolved in order to recognize expandable scalars in non-innermost loops. Until this generalization occurs, scalars like x in the preceding code segment inhibit parallelization of loops in which they are assigned. In the following example, scalar k is not expandable, and it is not an accumulator for a reduction.

```
    k = 1
     do i = 1, n
       do j = 1, n
1       a(j,i) = b(k) * x
       enddo
       k = i
2      if (i .gt. n/2) k = n - (i - n/2)
     enddo
```

If the outer loop is parallelized, conflicting values are stored into k by the various processors. The variable k cannot be made local to each processor because its value must remain coherent among the processors. It is possible the loop could be parallelized if all assignments to k are placed in critical sections. However, it is not clear where critical sections should be introduced because in general the value for k could depend on another scalar (or on k itself), and code to obtain the value of other scalars must reside in the same critical section.

In the previous example, the assignment to k within a conditional at label 2 prevents k from being recognized as an induction variable. If the conditional statement at label 2 is removed, k would be an induction variable whose value varies linearly with j, and the loop could be parallelized.

Scalar Last Values

During parallelization, scalars within loops often need to be privatized; that is, each execution thread has its own independent copy of the scalar. Problems can arise if a privatized scalar is accessed outside the loop. For example, consider the following loop:

```
for (i = 1; i<N; i++){
 if( f(x[i]) > 5.0 ) t = x[i];
}
 v = t;
```

The value of t may not be computed on the last iteration of the loop. Normally, if a scalar is assigned within a loop and used following the loop, the PGI compilers save the last value of the scalar. However, if the loop is parallelized and the scalar is not assigned on every iteration, it may be difficult, without resorting to costly critical sections, to determine on what iteration t is last assigned. Analysis allows the compiler to determine that a scalar is assigned on each iteration and hence that the loop is safe to parallelize if the scalar is used later, as illustrated in the following example.

```
for ( i = 1; i < n; i++) {
 if ( x[i] > 0.0 ) {
 t = 2.0;
 }
 else {
 t = 3.0;
 y[i] = ...t;
 }
```

```
}
v = t;
```

where `t` is assigned on every iteration of the loop. However, there are cases where a scalar may be privatizable, but if it is used after the loop, it is unsafe to parallelize. Examine the following loop in which each use of `t` within the loop is reached by a definition from the same iteration.

```
for ( i = 1; i < N; i++ ){
 if( x[i] > 0.0 ){
 t = x[i];
 ...
 ...
 y[i] = ...t;
 }
 }
 v = t;
```

Here `t` is privatizable, but the use of `t` outside the loop may yield incorrect results, since the compiler may not be able to detect on which iteration of the parallelized loop `t` is last assigned. The compiler detects the previous cases. When a scalar is used after the loop but is not defined on every iteration of the loop, parallelization does not occur.

When the programmer knows that the scalar is assigned on the last iteration of the loop, the programmer may use a directive or pragma to let the compiler know the loop is safe to parallelize. The Fortran directive `safe_lastval` informs the compiler that, for a given loop, all scalars are assigned in the last iteration of the loop; thus, it is safe to parallelize the loop. We could add the following line to any of our previous examples.

```
cpgi$l safe_lastval
```

The resulting code looks similar to this:

```
cpgi$l safe_lastval
...
for (i = 1; i<N; i++){
 if( f(x[i]) > 5.0 ) t = x[i];
}
 v = t;
```

In addition, a command-line option `–Msafe_lastval`, provides this information for all loops within the routines being compiled, which essentially provides global scope.

Processor-Specific Optimization & the Unified Binary

Different processors have differences, some subtle, in hardware features such as instruction sets and cache size. The compilers make architecture-specific decisions about things such as instruction selection, instruction scheduling, and vectorization. By default, the PGI compilers produce code specifically targeted to the type of processor on which the compilation is performed. That is, the default is to use all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

All PGI compilers have the capability of generating *unified binaries*, which provide a low-overhead means for generating a single executable that is compatible with and has good performance on more than one hardware platform.

You can use the `-tp` option to control compilation behavior by specifying the processor or processors with which the generated code is compatible. The compilers generate and combine into one executable multiple binary code streams, each optimized for a specific platform. At runtime, the one executable senses the environment and dynamically selects the appropriate code stream. For specific information on the `-tp` option, refer to –tp <target> [,target...].

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of from 10% to 90% compared to generating full copies of code for each target.

Programs can use the PGI Unified Binary even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mipa` option disables generation of PGI Unified Binary. Instead, the default target auto-detect rules for the host are used to select the target processor.

Interprocedural Analysis and Optimization using –Mipa

The PGI Fortran, C and C++ compilers use interprocedural analysis (IPA) that results in minimal changes to makefiles and the standard edit-build-run application development cycle. Other than adding `-Mipa` to the command line, no other changes are required. For reference and background, the process of building a program without IPA is described later in this section, followed by the minor modifications required to use IPA with the PGI compilers. While the PGCC compiler is used here to show how IPA works, similar capabilities apply to each of the PGI Fortran, C and C++ compilers.

Note

The examples use Linux file naming conventions. On Windows, '.o' files would be '.obj' files, and 'a.out' files would be '.exe' files.

Building a Program Without IPA – Single Step

Using the pgcc command-level compiler driver, multiple source files can be compiled and linked into a single executable with one command. The following example compiles and links three source files:

```
% pgcc -o a.out file1.c file2.c file3.c
```

In actuality, the pgcc driver executes several steps to produce the assembly code and object files corresponding to each source file, and subsequently to link the object files together into a single executable file. This command is roughly equivalent to the following commands performed individually:

```
% pgcc -S -o file1.s file1.c
% as -o file1.o file1.s
% pgcc -S -o file2.s file2.c
% as -o file2.o file2.s
% pgcc -S -o file3.s file3.c
% as -o file3.o file3.s
% pgcc -o a.out file1.o file2.o file3.o
```

If any of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgcc -o a.out file1.c file2.c file3.c
```

Note

This always works as intended, but has the side-effect of recompiling all of the source files, even if only one has changed. For applications with a large number of source files, this can be time-consuming and inefficient.

Building a Program Without IPA - Several Steps

It is also possible to use individual pgcc commands to compile each source file into a corresponding object file, and one to link the resulting object files into an executable:

```
% pgcc -c file1.c
% pgcc -c file2.c
% pgcc -c file3.c
% pgcc -o a.out file1.o file2.o file3.o
```

The pgcc driver invokes the compiler and assembler as required to process each source file, and invokes the linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgcc -c file1.c
% pgcc -o a.out file1.o file2.o file3.o
```

Building a Program Without IPA Using Make

The program compilation and linking process can be simplified greatly using the `make` utility on systems where it is supported. Suppose you create a `makefile` containing the following lines:

```
a.out: file1.o file2.o file3.o
 pgcc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
 pgcc $(OPT) -c file1.c
file2.o: file2.c
 pgcc $(OPT) -c file2.c
file3.o: file3.c
 pgcc $(OPT) -c file3.c
```

It is then possible to type a single make command:

```
% make
```

The `make` utility determines which object files are out of date with respect to their corresponding source files, and invokes the compiler to recompile only those source files and to relink the executable. If you subsequently edit one or more source files, the executable can be rebuilt with the minimum number of recompilations using the same single `make` command.

Building a Program with IPA

Interprocedural analysis and optimization (IPA) by the PGI compilers alters the standard and `make` utility command-level interfaces as little as possible. IPA occurs in three phases:

- **Collection**: Create a summary of each function or procedure, collecting the useful information for interprocedural optimizations. This is done during the compile step if the `–Mipa` switch is present on the command line; summary information is collected and stored in the object file.
- **Propagation**: Process all the object files to propagate the interprocedural summary information across function and file boundaries. This is done during the link step, when all the object files are combined, if the `–Mipa` switch is present on the link command line.
- **Recompile/Optimization**: Recompile each of the object files with the propagated interprocedural information, producing a specialized object file. This process is also performed during the link step when the `–Mipa` switch is present on the link command line.

When linking with `–Mipa`, the PGI compilers automatically regenerate IPA-optimized versions of each object file, essentially recompiling each file. If there are IPA-optimized objects from a previous build, the compilers will minimize the recompile time by reusing those objects if they are still valid. They will still be valid if the IPA-optimized object is newer than the original object file, and the propagated IPA information for that file has not changed since it was optimized.

After each object file has been recompiled, the regular linker is invoked to build the application with the IPA-optimized object files. The IPA-optimized object files are saved in the same directory as the original object files, for use in subsequent program builds.

Building a Program with IPA - Single Step

By adding the `–Mipa` command line switch, several source files can be compiled and linked with interprocedural optimizations with one command:

```
% pgcc -Mipa=fast -o a.out file1.c file2.c file3.c
```

Just like compiling without `–Mipa`, the driver executes several steps to produce the assembly and object files to create the executable:

```
% pgcc -Mipa=fast -S -o file1.s file1.c
% as -o file1.o file1.s
% pgcc -Mipa=fast -S -o file2.s file2.c
% as -o file2.o file2.s
% pgcc -Mipa=fast -S -o file3.s file3.c
% as -o file3.o file3.s
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

In the last step, an IPA linker is invoked to read all the IPA summary information and perform the interprocedural propagation. The IPA linker reinvokes the compiler on each of the object files to recompile them with interprocedural information. This creates three new objects with mangled names:

```
file1_ipa5_a.out.oo.o, file2_ipa5_a.out.oo.o, file2_ipa5_a.out.oo.o
```

The system linker is then invoked to link these IPA-optimized objects into the final executable. Later, if one of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgcc -Mipa=fast -o a.out file1.c file2.c file3.c
```

This will work, but again has the side-effect of compiling each source file, and recompiling each object file at link time.

Building a Program with IPA - Several Steps

Just by adding the `-Mipa` command-line switch, it is possible to use individual pgcc commands to compile each source file, followed by a command to link the resulting object files into an executable:

```
% pgcc -Mipa=fast -c file1.c
% pgcc -Mipa=fast -c file2.c
% pgcc -Mipa=fast -c file3.c
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

The pgcc driver invokes the compiler and assembler as required to process each source file, and invokes the IPA linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgcc -Mipa=fast -c file1.c
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

When the IPA linker is invoked, it will determine that the IPA-optimized object for `file1.o` (`file1_ipa5_a.out.oo.o`) is stale, since it is older than the object file1.o, and hence will need to be rebuilt, and will reinvoke the compiler to generate it. In addition, depending on the nature of the changes to the source file file1.c, the interprocedural optimizations previously performed for file2 and file3 may now be inaccurate. For instance, IPA may have propagated a constant argument value in a call from a function in `file1.c` to a function in `file2.c`; if the value of the argument has changed, any optimizations based on that constant value are invalid. The IPA linker will determine which, if any, of any previously created IPA-optimized objects need to be regenerated, and will reinvoke the compiler as appropriate to regenerate them. Only those objects that are stale or which have new or different IPA information will be regenerated, which saves on compile time.

Building a Program with IPA Using Make

As in the previous two sections, programs can be built with IPA using the make utility. Just add the command-line switch `-Mipa`, as shown here:

```
OPT=-Mipa=fast a.out: file1.o file2.o file3.o
 pgcc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
 pgcc $(OPT) -c file1.c
file2.o: file2.c
 pgcc $(OPT) -c file2.c
file3.o: file3.c
 pgcc $(OPT) -c file3.c
```

Using the single `make` command invokes the compiler to generate any object files that are out-of-date, then invokes pgcc to link the objects into the executable; at link time, pgcc calls the IPA linker to regenerate any stale or invalid IPA-optimized objects.

```
% make
```

Questions about IPA

1. Why is the object file so large?

An object file created with `–Mipa` contains several additional sections. One is the summary information used to drive the interprocedural analysis. In addition, the object file contains the compiler internal representation of the source file, so the file can be recompiled at link time with interprocedural optimizations. There may be additional information when inlining is enabled. The total size of the object file may be 5-10 times its original size. The extra sections are not added to the final executable.

2. What if I compile with `–Mipa` and link without `–Mipa`?

The PGI compilers generate a legal object file, even when the source file is compiled with `–Mipa`. If you compile with `–Mipa` and link without `–Mipa`, the linker is invoked on the original object files. A legal executable will be generated; while this will not have the benefit of interprocedural optimizations, any other optimizations will apply.

3. What if I compile without `–Mipa` and link with `–Mipa`?

At link time, the IPA linker must have summary information about all the functions or routines used in the program. This information is created only when a file is compiled with `–Mipa`. If you compile a file without `–Mipa` and then try to get interprocedural optimizations by linking with `–Mipa`, the IPA linker will issue a message that some routines have no IPA summary information, and will proceed to run the system linker using the original object files. If some files were compiled with `–Mipa` and others were not, it will determine the safest approximation of the IPA summary information for those files not compiled with `–Mipa`, and use that to recompile the other files using interprocedural optimizations.

4. Can I build multiple applications in the same directory with `–Mipa`?

Yes. Suppose you have three source files: `main1.c`, `main2.c`, and `sub.c`, where `sub.c` is shared between the two applications. Suppose you build the first application with `–Mipa`, using this command:

```
% pgcc -Mipa=fast -o app1 main1.c sub.c
```

The the IPA linker creates two IPA-optimized object files:

```
main1_ipa4_app1.o sub_ipa4_app1.oo
```

It uses them to build the first application. Now suppose you build the second application using this command:

```
% pgcc -Mipa=fast -o app2 main2.c sub.c
```

The IPA linker creates two more IPA-optimized object files:

```
main2_ipa4_app2.oo sub_ipa4_app2.oo
```

Note

There are now three object files for `sub.c`: the original `sub.o`, and two IPA-optimized objects, one for each application in which it appears.

5. How is the mangled name for the IPA-optimized object files generated?

The mangled name has '_ipa' appended, followed by the decimal number of the length of the executable file name, followed by an underscore and the executable file name itself. The suffix is changed to .oo (on Linux or Mac OS X) or .oobj (on Windows) so linking *.o or *.obj does not pull in the IPA-optimized objects. If the IPA linker determines that the file would not benefit from any

interprocedural optimizations, it does not have to recompile the file at link time and uses the original object.

Profile-Feedback Optimization using –Mpfi/–Mpfo

The PGI compilers support many common profile-feedback optimizations, including semi-invariant value optimizations and block placement. These are performed under control of the `-Mpfi`/`-Mpfo` command-line options.

When invoked with the `-Mpfi` option, the PGI compilers instrument the generated executable for collection of profile and data feedback information. This information can be used in subsequent compilations that include the `-Mpfo` optimization option. `-Mpfi` must be used at both compile-time and link-time. Programs compiled with `-Mpfi` include extra code to collect runtime statistics and write them out to a trace file. When the resulting program is executed, a profile feedback trace file `pgfi.out` is generated in the current working directory.

Note

Programs compiled and linked with `-Mpfi` execute more slowly due to the instrumentation and data collection overhead. You should use executables compiled with `-Mpfi` only for execution of training runs.

When invoked with the `-Mpfo` option, the PGI compilers use data from a `pgfi.out` profile feedback tracefile to enable or enhance certain performance optimizations. Use of this option requires the presence of a `pgfi.out` trace file in the current working directory.

Default Optimization Levels

The following table shows the interaction between the –O<level>, –g, and `-M<opt>` options. In the table, level can be 0, 1, 2, 3 or 4, and <opt> can be `vect`, `concur`, `unroll` or `ipa`. The default optimization level is dependent upon these command-line options.

Table 3.1. Optimization and –O, –g and –M<opt> Options

Optimize Option	Debug Option	–M<opt> Option	Optimization Level
none	none	none	1
none	none	–M<opt>	2
none	–g	none	0
–O	none or –g	none	2
–Olevel	none or –g	none	level
–Olevel <= 2	none or –g	–M<opt>	2

Code that is not optimized yet compiled using the option `-O0` can be significantly slower than code generated at other optimization levels. The `-M<opt>` option, where <opt> is `vect`, `concur`, `unroll` or `ipa`, sets the optimization level to 2 if no `-O` options are supplied. Both the `-fast` and the `-fastsse` options set the optimization level to a target-dependent optimization level if no `-O` options are supplied.

Local Optimization Using Directives and Pragmas

Command-line options let you specify optimizations for an entire source file. Directives supplied within a Fortran source file and pragmas supplied within a C or C++ source file provide information to the compiler and alter the effects of certain command-line options or the default behavior of the compiler. (Many directives have a corresponding command-line option.)

While a command line option affects the entire source file that is being compiled, directives and pragmas let you do the following:

- Apply, or disable, the effects of a particular command-line option to selected subprograms or to selected loops in the source file (for example, an optimization).
- Globally override command-line options.
- Tune selected routines or loops based on your knowledge or on information obtained through profiling.

Chapter 9, "*Using Directives and Pragmas*" provides details on how to add directives and pragmas to your source files.

Execution Timing and Instruction Counting

As this chapter describes, once you have a program that compiles, executes and gives correct results, you may optimize your code for execution efficiency.

Selecting the correct optimization level requires some thought and may require that you compare several optimization levels before arriving at the best solution. To compare optimization levels, you need to measure the execution time for your program. There are several approaches you can take for timing execution.

- You can use shell commands that provide execution time statistics.
- You can include function calls in your code that provide timing information.
- You can profile sections of code.

Timing functions available with the PGI compilers include these:

- 3F timing routines
- The SECNDS pre-declared function in PGF77, PGF95, or PGFORTRAN
- The SYSTEM_CLOCK or CPU_CLOCK intrinsics in PGF95.

In general, when timing a program, you should try to eliminate or reduce the amount of system level activities such as I/O, program loading, and task switching.

The following example shows a *fragment* that indicates how to use SYSTEM_CLOCK effectively within an F90 or F95 program unit.

Example 3.4. Using SYSTEM_CLOCK code fragment

```
integer :: nprocs, hz, clock0, clock1
 real :: time
!
call system_clock (count_rate=hz)
!
 call system_clock(count=clock0)
 < do work>
 call system_clock(count=clock1)
!
 t = (clock1 - clock0)
 time = real (t) / real(hz)
```

Or you can use the F90 `cpu_time` subroutine:

```
real :: t1, t2, time
call cpu_time(t1)
 < do work>
call cpu_time(t2)
time = t2 - t1
```

Portability of Multi-Threaded Programs on Linux

PGI created the library `libnuma` to handle the variations between various implementations of Linux.

Some older versions of Linux are lacking certain features that support multi-processor and multi-core systems; in particular, the system call 'sched_setaffinity' and the numa library `libnuma`. The PGI runtime library uses these features to implement some `-Mconcur` and `-mp` operations.

These variations led to the creation of the PGI library: `libnuma`, which is used on all 32-bit and 64-bit Linux systems, but is not needed on Windows or Mac OS X.

When a program is linked with the system `libnuma` library, the program depends on that library to run. On systems without a system `libnuma` library, the PGI version of `libnuma` provides the required stubs so that the program links and executes properly. If the program is linked with `libnuma`, the differences between systems is masked by the different versions of `libnuma`.

When a program is deployed to the target system, the proper set of libraries, real or stub, should be deployed with the program.

This facility requires that the program be dynamically linked with `libnuma`.

libnuma

Not all systems have libnuma. Typically, only numa systems have this library. PGI supplies a stub version of libnuma which satisfies the calls from the PGI runtime to libnuma. libnuma is a shared library that is linked dynamically at runtime.

The reason to have a numa library on all systems is to allow multi-threaded programs, such as programs compiled with `-Mconcur` or `-mp`, to be compiled, linked, and executed without regard to whether the host or

target systems has a numa library. When the numa library is not available, a multi-threaded program still runs because the calls to the numa library are satisfied by the PGI stub library.

During installation, the installation procedure checks for the existence of a real libnuma among the system libraries. If the real library is not found, the PGI stub version is substituted.

Chapter 4. Using Function Inlining

Function inlining replaces a call to a function or a subroutine with the body of the function or subroutine. This can speed up execution by eliminating parameter passing and function/subroutine call and return overhead. It also allows the compiler to optimize the function with the rest of the code. Note that using function inlining indiscriminately can result in much larger code size and no increase in execution speed.

The PGI compilers provide two categories of inlining:

- **Automatic inlining** - During the compilation process, a hidden pass precedes the compilation pass. This hidden pass extracts functions that are candidates for inlining. The inlining of functions occurs as the source files are compiled.
- **Inline libraries** - You create inline libraries, for example using the pgfortran compiler driver and the `-o` and `-Mextract` options. There is no hidden extract pass but you must ensure that any files that depend on the inline library use the latest version of the inline library.

There are important restrictions on inlining. Inlining only applies to certain types of functions. Refer to "Restrictions on Inlining," on page 52 for more details on function inlining limitations.

This chapter describes how to use the following options related to function inlining:

```
-Mextract
-Minline
-Mrecursive
```

Invoking Function Inlining

To invoke the function inliner, use the `-Minline` option. If you do not specify an inline library, the compiler performs a special prepass on all source files named on the compiler command line before it compiles any of them. This pass extracts functions that meet the requirements for inlining and puts them in a temporary inline library for use by the compilation pass.

Several `-Minline` suboptions let you determine the selection criteria for functions to be inlined. These suboptions include:

except:`func`
: Inlines all eligible functions except `func`, a function in the source text. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Inlines all functions in the source text whose name matches `func`. you can use a comma-separated list to specify multiple functions.

[size:]`n`
: Inlines functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size `n` may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

levels:`n`
: Inlines `n` level of function calling levels. The default number is one (1). Using a level greater than one indicates that function calls within inlined functions may be replaced with inlined code. This approach allows the function inliner to automatically perform a sequence of inline and extract processes.

[lib:]`file.ext`
: Instructs the inliner to inline the functions within the library file `file.ext`. If no inline library is specified, functions are extracted from a temporary library created during an extract prepass.

> **Tip**
>
> Create the library file using the `-Mextract` option.

If you specify both a function name and a size n, the compiler inlines functions that match the function name *or* have n or fewer statements.

If a name is used without a keyword, then a name with a period is assumed to be an inline library and a name without a period is assumed to be a function name. If a number is used without a keyword, the number is assumed to be a size.

In the following example, the compiler inlines functions with fewer than approximately 100 statements in the source file `myprog.f` and writes the executable code in the default output file `a.out`.

```
$ pgfortran -Minline=size:100 myprog.f
```

Refer to "–M Options by Category" in the PGI Compiler Reference Manual.

For more information on the `-Minline` options, refer to "–M Options by Category" chapter of the PGI Compiler Reference Manual.

Using an Inline Library

If you specify one or more inline libraries on the command line with the `-Minline` option, the compiler does not perform an initial extract pass. The compiler selects functions to inline from the specified inline library.

If you also specify a size or function name, all functions in the inline library meeting the selection criteria are selected for inline expansion at points in the source text where they are called.

If you do not specify a function name or a size limitation for the `-Minline` option, the compiler inlines every function in the inline library that matches a function in the source text.

In the following example, the compiler inlines the function `proc` from the inline library `lib.il` and writes the executable code in the default output file `a.out`.

```
$ pgfortran -Minline=name:proc,lib:lib.il myprog.f
```

The following command line is equivalent to the preceding line, with the exception that in the following example does not use the keywords `name:` and `lib:`. You typically use keywords to avoid name conflicts when you use an inline library name that does not contain a period. Otherwise, without the keywords, a period informs the compiler that the file on the command line is an inline library.

```
$ pgfortran -Minline=proc,lib.il myprog.f
```

Creating an Inline Library

You can create or update an inline library using the `-Mextract` command-line option. If you do not specify selection criteria with the `-Mextract` option, the compiler attempts to extract all subprograms.

Several `-Mextract` options let you determine the selection criteria for creating or updating an inline library. These selection criteria include:

`func`
: Extracts the function `func`. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Extracts the functions whose name matches `func`, a function in the source text.

[size:]`n`
: Limits the size of the extracted functions to functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size n may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

[lib:]`ext.lib`
: Stores the extracted information in the library directory `ext.lib`.

  If no inline library is specified, functions are extracted to a temporary library created during an extract prepass for use during the compilation stage.

When you use the `-Mextract` option, only the extract phase is performed; the compile and link phases are not performed. The output of an extract pass is a library of functions available for inlining. This output is placed in the inline library file specified on the command line with the `–o` filename specification. If the library

file exists, new information is appended to it. If the file does not exist, it is created. You can use a command similar to the following:

```
$ pgfortran -Mextract=lib:lib.il myfunc.f
```

You can use the `-Minline` option with the `-Mextract` option. In this case, the extracted library of functions can have other functions inlined into the library. Using both options enables you to obtain more than one level of inlining. In this situation, if you do not specify a library with the `–Minline` option, the inline process consists of two extract passes. The first pass is a hidden pass implied by the `–Minline` option, during which the compiler extracts functions and places them into a temporary library. The second pass uses the results of the first pass but puts its results into the library that you specify with the `–o` option.

Working with Inline Libraries

An inline library is implemented as a directory with each inline function in the library stored as a file using an encoded form of the inlinable function.

A special file named `TOC` in the inline library directory serves as a table of contents for the inline library. This is a printable, ASCII file which you can examine to locate information about the library contents, such as names and sizes of functions, the source file from which they were extracted, the version number of the extractor which created the entry, and so on.

Libraries and their elements can be manipulated using ordinary system commands.

- Inline libraries can be copied or renamed.
- Elements of libraries can be deleted or copied from one library to another.
- The `ls` or `dir` command can be used to determine the last-change date of a library entry.

Dependencies

When a library is created or updated using one of the PGI compilers, the last-change date of the library directory is updated. This allows a library to be listed as a dependence in a makefile and ensures that the necessary compilations are performed when a library is changed.

Updating Inline Libraries - Makefiles

If you use inline libraries you must be certain that they remain up-to-date with the source files into which they are inlined. One way to assure inline libraries are updated is to include them in a makefile.

The makefile fragment in the following example assumes the file `utils.f` contains a number of small functions used in the files `parser.f` and `alloc.f`.

This portion of the makefile:

- Maintains the inline library `utils.il`.
- Updates the library whenever you change `utils.f` or one of the include files it uses.
- Compiles `parser.f` and `alloc.f` whenever you update the library.

Example 4.1. Sample Makefile

```
SRC = mydir
FC = pgfortran
FFLAGS = -O2
main.o: $(SRC)/main.f $(SRC)/global.h
 $(FC) $(FFLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
 $(FC) $(FFLAGS) -c $(SRC)/utils.f
utils.il: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
 $(FC) $(FFLAGS) -Mextract=15 -o utils.il utils.f
parser.o: $(SRC)/parser.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/alloc.f
myprog: main.o utils.o parser.o alloc.o
 $(FC) -o myprog main.o utils.o parser.o alloc.o
```

Error Detection during Inlining

You can specify the `–Minfo=inline` option to request inlining information from the compiler when you invoke the inliner. For example:

```
$ pgfortran -Minline=mylib.il -Minfo=inline myext.f
```

Examples

Assume the program `dhry` consists of a single source file `dhry.f`. The following command line builds an executable file for `dhry` in which proc7 is inlined wherever it is called:

```
$ pgfortran dhry.f -Minline=proc7
```

The following command lines build an executable file for `dhry` in which proc7 plus any functions of approximately 10 or fewer statements are inlined (one level only).

> **Note**
>
> The specified functions are inlined only if they are previously placed in the inline library, `temp.il`, during the extract phase.

```
$ pgfortran dhry.f -Mextract=lib:temp.il
$ pgfortran dhry.f -Minline=10,proc7,temp.il
```

Using the same source file `dhry.f`, the following example builds an executable for `dhry` in which all functions of roughly ten or fewer statements are inlined. Two levels of inlining are performed. This means that if function A calls function B, and B calls C, and both B and C are inlinable, then the version of B which is inlined into A will have had C inlined into it.

```
$ pgfortran dhry.f -Minline=size:10,levels:2
```

Restrictions on Inlining

The following Fortran subprograms cannot be extracted:

- Main or BLOCK DATA programs.
- Subprograms containing alternate return, assigned GO TO, DATA, SAVE, or EQUIVALENCE statements.
- Subprograms containing FORMAT statements.
- Subprograms containing multiple entries.

A Fortran subprogram is not inlined if any of the following applies:

- It is referenced in a statement function.
- A common block mismatch exists; in other words, the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- An argument mismatch exists; in other words, the number and type (size) of actual and formal parameters must be equal.
- A name clash exists, such as a call to subroutine `xyz` in the extracted subprogram and a variable named `xyz` in the caller.

The following types of C and C++ functions cannot be inlined:

- Functions containing switch statements
- Functions which reference a static variable whose definition is nested within the function
- Function which accept a variable number of arguments

Certain C/C++ functions can only be inlined into the file that contains their definition:

- Static functions
- Functions which call a static function
- Functions which reference a static variable

Chapter 5. Using OpenMP

The PGF77, PGF95, and PGFORTRAN Fortran compilers support the OpenMP Fortran Application Program Interface. The PGCC ANSI C and C++ compilers support the OpenMP C/C++ Application Program Interface.

OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs. OpenMP may be used to obtain most of the parallel performance you can expect from your code, or it may serve as a stepping stone to parallelizing an entire application with MPI.

This chapter provides information on OpenMP as it is supported by PGI compilers.

Use the `-mp` compiler switch to enable processing of the OMP pragmas listed in this chapter. C++ applications also compile with thread-safe versions of STL header files. As of Release 2011, the `-mp` switch is enabled by default to link the OpenMP runtime library, and for C++, the thread-safe Standard Template Library. To disable this option, use `-nomp`.

Note

The C++ Standard Template library is thread-safe to the extent allowed in the STLport code: simultaneous accesses to distinct containers are safe, simultaneous reads to shared containers are also safe. However, simultaneous writes to shared containers must be protected by **#pragma omp critical** sections.

This chapter describes how to use the following option related to using OpenMP:

`-mp`

OpenMP Overview

Let's look at the OpenMP shared-memory parallel programming model and some common OpenMP terminology.

OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model is a collection of compiler directives or pragmas, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and in C/C++ programs.

Fortran directives and C/C++ pragmas
: Allow users to mark sections of code that can be executed in parallel when the code is compiled using the `-mp` switch. When this switch is not present, the compiler ignores these directives and pragmas.

  OpenMP Fortran directives begin with `!$OMP` , `C$OMP`, or `*$OMP`, beginning in column 1. OpenMP pragmas for C/C++ begin with `#pragma omp`. This format allows the user to have a single source for use with or without the `-mp` switch, as these lines are then merely viewed as comments when `-mp` is not present or the compilers are not capable of handling directives or C/C++ pragmas.

  These directives and pragmas allow the user to create task, loop, and parallel section work-sharing constructs and synchronization constructs. They also allow the user to define how data is shared or copied between parallel threads of execution.

  Fortran directives and C/C++ pragmas include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations or C/C++ for loop iterations should be split among the available threads of execution, and synchronization constructs.

  Note

  The data environment is controlled either by using clauses on the directives or pragmas, or with additional directives or pragmas.

Runtime library routines
: Are available to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region.

Environment variables
: Are available to control the execution behavior of parallel programs. For more information on OpenMP, see *www.openmp.org*.

Macro substitution
: C and C++ omp pragmas are subject to macro replacement after `#pragma omp`.

Terminology

For OpenMP 3.1 there are a number of terms for which it is useful to have common definitions.

Thread
: An execution entity with a stack and associated static memory, called *threadprivate memory*.

  - An OpenMP thread is a thread that is managed by the OpenMP runtime system.
  - A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

Region
: All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

  Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. PGI supports non-lexically nested parallel regions.

Parallel region

In OpenMP 3.1 there is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

- An inactive parallel region is executed by a single thread.
- An active parallel region is a parallel region that is executed by a team consisting of more than one thread.

Note

The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.1. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

```
     program test
     logical omp_in_parallel

!$omp parallel
     print *, omp_in_parallel()
!$omp end parallel

     stop
     end
```

Suppose we run this program with OMP_NUM_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.1, the program yields F. In OpenMP 3.1, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

Task

A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

OpenMP Example

Look at the following simple OpenMP example involving loops.

Example 5.1. OpenMP Loop Example

```
      PROGRAM MAIN
      INTEGER I, N, OMP_GET_THREAD_NUM
      REAL*8 V(1000), GSUM, LSUM
      GSUM = 0.0D0
      N = 1000
      DO I = 1, N
         V(I) = DBLE(I)
      ENDDO
```

```
!$OMP PARALLEL PRIVATE(I,LSUM) SHARED(V,GSUM,N)
      LSUM = 0.0D0
!$OMP DO
      DO I = 1, N
         LSUM = LSUM + V(I)
      ENDDO
!$OMP END DO
!$OMP CRITICAL
```

```
      print *, "Thread ",OMP_GET_THREAD_NUM()," local sum: ",LSUM
      GSUM = GSUM + LSUM
!$OMP END CRITICAL
!$OMP END PARALLEL

      PRINT *, "Global Sum: ",GSUM
      STOP
      END
```

If you execute this example with the environment variable OMP_NUM_THREADS set to 4, then the output looks similar to this:

```
Thread            0 local sum:    31375.00000000000
Thread            1 local sum:    93875.00000000000
Thread            2 local sum:    156375.0000000000
Thread            3 local sum:    218875.0000000000
Global Sum:    500500.0000000000
FORTRAN STOP
```

Task Overview

Every part of an OpenMP program is part of a task. A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

In the following sections, we use this terminology:

Task
: The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

  - An explicit task is a task generated when a task construct is encountered during execution.
  - An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

Task construct
: A task directive plus a structured block

Task region
: The dynamic sequence of instructions produced by the execution of a task by a thread.

Fortran Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the PGI Fortran compilers when the option `–mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

With the exception of the SGI-compatible DOACROSS directive, the *sentinel* must comply with these rules:

- Be one of these: !OMP, COMP, or *$OMP.
- Must start in column 1 (one).
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is C$.

The *directive_name* can be any of the directives listed in Table 5.1, "Directive and Pragma Summary Table". The valid clauses depend on the directive. Table 5.2, "Directive and Pragma Clauses Summary Table " provides a list of clauses, the directives to which they apply., and their functionality.

In addition to the sentinel rules, the directive must also comply with these rules:

- Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- Initial directive lines must have a space or zero in column six.
- Continuation directive lines must have a character other than a space or a zero in column six. Continuation lines for C$DOACROSS directives are specified using the C$& sentinel.
- Directives which are presented in pairs must be used in pairs.

Clauses associated with directives have these characteristics:

- The order in which clauses appear in the parallelization directives is not significant.
- Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.
- Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

C/C++ Parallelization Pragmas

Parallelization pragmas are #pragma statements in a C or C++ program that are interpreted by the PGCC C and C++ compilers when the option -mp is specified on the command line. The form of a parallelization pragma is:

```
#pragma omp pragma_name [clauses]
```

The format for pragmas include these standards:

- The pragmas follow the conventions of the C and C++ standards.
- Whitespace can appear before and after the #.
- Preprocessing tokens following the #pragma omp are subject to macro replacement.
- The order in which clauses appear in the parallelization pragmas is not significant.
- Spaces separate clauses within the pragmas.

- Clauses on pragmas may be repeated as needed subject to the restrictions listed in the description of each clause.

For the purposes of the OpenMP pragmas, a C/C++ structured block is defined to be a statement or compound statement (a sequence of statements beginning with { and ending with }) that has a single entry and a single exit. No statement or compound statement is a C/C++ structured block if there is a jump into or out of that statement.

Directive and Pragma Recognition

The compiler option `-mp` enables recognition of the parallelization directives and pragmas. The use of this option also implies:

`-Mreentrant`
: Local variables are placed on the stack and optimizations, such as `-Mnoframe`, that may result in non-reentrant code are disabled.

`-Miomutex`
: For directives, critical sections are generated around Fortran I/O statements.

For pragmas, calls to I/O library functions are system-dependent and are not necessarily guaranteed to be thread-safe. I/O library calls within parallel regions should be protected by critical regions to ensure they function correctly on all systems.

Directive and Pragma Summary Table

The following table provides a brief summary of the directives and pragmas that PGI supports.

Note

In the table, the values in uppercase letters are Fortran directives while the names in lowercase letters are C/C++ pragmas.

Table 5.1. Directive and Pragma Summary Table

Fortran Directive and C/C++ Pragma	Description
ATOMIC ... END ATOMIC and atomic	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive or critical pragma. The END ATOMIC directive is only allowed when ending ATOMIC CAPTURE regions. Note: Only certain statements are allowed.
BARRIER and barrier	Synchronizes all threads at a specific point in a program so that all threads complete work to that point before any thread continues.
CRITICAL ... END CRITICAL and critical	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
DO...END DO and for	Provides a mechanism for distribution of loop iterations across the available threads in a parallel region.

Fortran Directive and C/C++ Pragma	Description
C$DOACROSS	Specifies that the compiler should parallelize the loop to which it applies, even though that loop is not contained within a parallel region.
FLUSH and flush	When this appears, all processor-visible data items, or, when a list is present (FLUSH [list]), only those specified in the list, are written to memory, thus ensuring that all the threads in a team have a consistent view of certain objects in memory.
MASTER ... END MASTER and master	Designates code that executes on the master thread and that is skipped by the other threads.
ORDERED and ordered	Defines a code block that is executed by only one thread at a time, and in the order of the loop iterations; this makes the ordered code block sequential, while allowing parallel execution of statements outside the code block.
PARALLEL DO and parallel for	Enables you to specify which loops the compiler should parallelize.
PARALLEL ... END PARALLEL and parallel	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
PARALLEL SECTIONS and parallel sections	Defines a non-iterative work-sharing construct without the need to define an enclosing parallel region.
PARALLEL WORKSHARE ... END PARALLEL WORKSHARE	Provides a short form method for including a WORKSHARE directive inside a PARALLEL construct.
SECTIONS … END SECTIONS and sections	Defines a non-iterative work-sharing construct within a parallel region.
SINGLE ... END SINGLE and single	Designates code that executes on a single thread and that is skipped by the other threads.
TASK and task	Defines an explicit task.
TASKYIELD and taskyield	Specifies a scheduling point for a task where the currently executing task may be yielded, and a different deferred task may be executed.
TASKWAIT and taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
THREADPRIVATE and threadprivate	When a common block or variable that is initialized appears in this directive or pragma, each thread's copy is initialized once prior to its first use.
WORKSHARE ... END WORKSHARE	Provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Directive and Pragma Clauses

Some directives and pragmas accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives, so the clauses that are valid are included with the description of the directive and pragma.

The following table provides a brief summary of the clauses associated with OPENMP directives and pragmas that PGI supports.

Table 5.2. Directive and Pragma Clauses Summary Table

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"CAPTURE"	ATOMIC	atomic	Specifies that the atomic action is reading and updating, or writing and updating a value, capturing the intermediate state.
"COLLAPSE (n)"	DO...END DO PARALLEL DO PARALLEL WORKSHARE	parallel for	Specifies how many loops are associated with the loop construct.
"COPYIN (list)"	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for	Allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region. Then, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.
"COPYPRIVATE(list)"	SINGLE	single	Specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.
"DEFAULT"	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.
"FINAL"	TASK	task	Specifies that all subtasks of this task will be run immediately.

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"FIRSTPRIVATE(list)"	DO PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	for parallel parallel for sections single	Specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.
"IF()"	PARALLEL ... END PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Specifies whether a loop should be executed in parallel or in serial.
"LASTPRIVATE(list)"	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	parallel parallel for parallel sections sections	Specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration of a for-loop construct or last section of #pragma sections.
"MERGEABLE"	TASK	task	Specifies that this task will run with the same data environment, including OpenMP internal control variables, as when it is encountered.
"NOWAIT"	DO ... END DO SECTIONS SINGLE WORKSHARE . . . END WORKSHARE	for sections single	Overrides the barrier implicit in a directive.
"NUM_THREADS"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Sets the number of threads in a thread team.
"ORDERED"	DO...END DO PARALLEL DO... END PARALLEL DO	parallel for	Required on a parallel FOR statement if an ordered directive is used in the loop.

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"PRIVATE"	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	for parallel parallel for parallel sections sections single	Specifies that each thread should have its own instance of a variable.
"READ"	ATOMIC	atomic	Specifies that the atomic action is reading a value.
"REDUCTION" `( {operator | intrinsic } : list)`	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS	for parallel parallel for parallel sections sections	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
"SCHEDULE" `(type[ ,chunk])`	DO ... END DO PARALLEL DO... END PARALLEL DO	for parallel for	Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.
"SHARED"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Specifies that one or more variables should be shared among all threads. All threads within a team access the same storage area for shared variables
"UNTIED"	TASK TASKWAIT	task taskwait	Specifies that any thread in the team can resume the task region after a suspension.
"UPDATE"	ATOMIC	atomic	Specifies that the atomic action is updating a value.
"WRITE"	ATOMIC	atomic	Specifies that the atomic action is writing a value.

For complete information on these clauses, refer to the OpenMP documentation available on the World Wide Web.

Runtime Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

Any C/C++ program unit that invokes these functions should include the statement #include <omp.h>. The `omp.h` include file contains definitions for each of the C/C++ library routines and the required type definitions. For example, to use the `omp_get_num_threads` function, use this syntax:

```
#include <omp.h>
int omp_get_num_threads(void);
```

Note

Unlimited OpenMP thread counts are available in all PGI configurations. The number of threads is unlicensed in the OpenMP runtime libraries - up to the hard limit of 64 threads.

The following table summarizes the runtime library calls.

Note

The Fortran call is shown first followed by the equivalent C/C++ call.

Table 5.3. Runtime Library Routines Summary

Runtime Library Routines with Examples	
omp_get_num_threads Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable `OMP_NUM_THREADS` or to the value set by the last previous call to **omp_set_num_threads()**.	
Fortran	`integer function omp_get_num_threads()`
C/C++	`int omp_get_num_threads(void);`

Runtime Library Routines with Examples	
omp_set_num_threads	
Sets the number of threads to use for the next parallel region. This subroutine or function can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine or function that is called from within a parallel region, the results are undefined. Further, this subroutine or function has precedence over the `OMP_NUM_THREADS` environment variable.	
Fortran	`subroutine omp_set_num_threads(scalar_integer_exp)`
C/C++	`void omp_set_num_threads(int num_threads);`
omp_get_thread_num	
Returns the thread number within the team. The thread number lies between 0 and **omp_get_num_threads()-1**. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.	
Fortran	`integer function omp_get_thread_num()`
C/C++	`int omp_get_thread_num(void);`
omp_get_ancestor_thread_num	
Returns, for a given nested level of the current thread, the thread number of the ancestor.	
Fortran	`integer function omp_get_ancestor_thread_num(level)` `integer level`
C/C++	`int omp_get_ancestor_thread_num(int level);`
omp_get_active_level	
Returns the number of enclosing active parallel regions enclosing the task that contains the call. PGI currently supports only one level of active parallel regions, so the return value currently is 1.	
Fortran	`integer function omp_get_active_level()`
C/C++	`int omp_get_active_level(void);`
omp_get_level	
Returns the number of parallel regions enclosing the task that contains the call.	
Fortran	`integer function omp_get_level()`
C/C++	`int omp_get_level(void);`

Runtime Library Routines with Examples	
omp_get_max_threads	
Returns the maximum value that can be returned by calls to **omp_get_num_threads()**. If **omp_set_num_threads()** is used to change the number of processors, subsequent calls to **omp_get_max_threads()** return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code.	
Fortran	`integer function omp_get_max_threads()`
C/C++	`void omp_get_max_threads(void);`
omp_get_num_procs	
Returns the number of processors that are available to the program	
Fortran	`integer function omp_get_num_procs()`
C/C++	`int omp_get_num_procs(void);`
omp_get_stack_size	
Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread. This value may *not* be the size of the stack of the current thread.	

Fortran	`!omp_get_stack_size interface`
	` function omp_get_stack_size ()`
	` use omp_lib_kinds`
	` integer ( kind=OMP_STACK_SIZE_KIND )`
	` :: omp_get_stack_size`
	` end function omp_get_stack_size`
	` end interface`

C/C++	`size_t omp_get_stack_size(void);`
omp_set_stack_size	
Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread. The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the PGI implementation, all OpenMP or auto-parallelization threads are created just prior to the first parallel region; therefore, only calls to `omp_set_stack_size()` that occur prior to the first region have an effect.	
Fortran	`subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND))`
C/C++	`void omp_set_stack_size(size_t);`
omp_get_team_size	
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.	
Fortran	`integer function omp_get_team_size (level)` `integer level`
C/C++	`integer omp_get_team_size(int level);`

Runtime Library Routines with Examples	
omp_in_final	
Returns whether or not we are within a final task. Returns `.TRUE.` for directives and non-zero for pragmas if called from within a parallel region and `.FALSE.` for directives and zero for pragmas if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.` for directives and zero for pragmas, the function returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`integer function omp_in_final()`
C/C++	`int omp_in_final(void);`
omp_in_parallel	
Returns whether or not the call is within a parallel region. Returns `.TRUE.` for directives and non-zero for pragmas if called from within a parallel region and `.FALSE.` for directives and zero for pragmas if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.` for directives and zero for pragmas, the function returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_in_parallel()`
C/C++	`int omp_in_parallel(void);`
omp_set_dynamic	
Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_dynamic(scalar_logical_exp)`
C/C++	`void omp_set_dynamic(int dynamic_threads);`
omp_get_dynamic	
Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_get_dynamic()`
C/C++	`void omp_get_dynamic(void);`
omp_set_nested	
Allows enabling/disabling of nested parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_nested(nested)` `  logical nested`
C/C++	`void omp_set_nested(int nested);`

Runtime Library Routines with Examples	
omp_get_nested	
Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_get_nested()`
C/C++	`int omp_get_nested(void);`
omp_set_schedule	
Set the value of the run_sched_var.	

Fortran	`subroutine omp_set_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

C/C++	`double omp_set_schedule()`
omp_get_schedule	
Retrieve the value of the run_sched_var.	

Fortran	`subroutine omp_get_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

C/C++	`double omp_get_schedule()`
omp_get_wtime	
Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value for directives and as a floating-point double value for pragmas. Times returned are per-thread times, and are not necessarily globally consistent across all threads.	
Fortran	`double precision function omp_get_wtime()`
C/C++	`double omp_get_wtime(void)`
omp_get_wtick	
Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value for Fortran directives and as a floating-point double value for C/C++ pragmas.	
Fortran	`double precision function omp_get_wtick()`
C/C++	`double omp_get_wtick();`

Runtime Library Routines with Examples	
omp_init_lock	
Initializes a lock associated with the variable lock for use in subsequent calls to lock routines. The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_init_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_init_lock(omp_lock_t *lock);` `void omp_init_nest_lock(omp_nest_lock_t *lock);`
omp_destroy_lock	
Disassociates a lock associated with the variable.	

Fortran	`subroutine omp_destroy_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_destroy_lock(omp_lock_t *lock);` `void omp_destroy_nest_lock(omp_nest_lock_t *lock);`
omp_set_lock	
Causes the calling thread to wait until the specified lock is available. The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_set_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_set_lock(omp_lock_t *lock);` `void omp_set_nest_lock(omp_nest_lock_t *lock);`
omp_unset_lock	
Causes the calling thread to release ownership of the lock associated with `integer_var`. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_unset_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`#include <omp.h> void omp_unset_lock(omp_lock_t *lock);` `void omp_unset_nest_lock(omp_nest_lock_t *lock);`
omp_test_lock	
Causes the calling thread to try to gain ownership of the lock associated with the variable. The function returns `.TRUE.` for directives and non-zero for pragmas if the thread gains ownership of the lock; otherwise it returns `.FALSE.` for directives and zero for pragmas. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`logical function omp_test_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

Runtime Library Routines with Examples	
C/C++	`int omp_test_lock(omp_lock_t *lock);` `int omp_test_nest_lock(omp_nest_lock_t *lock);`

Environment Variables

You can use OpenMP environment variables to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives and pragmas.

The following summary table is a quick reference for the OpenMP environment variables that PGI uses.

Table 5.4. OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_MAX_ACTIVE_LEVELS		Specifies the maximum number of nested parallel regions.
OMP_NESTED	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) nested parallelism.
OMP_NUM_THREADS	1	Specifies the number of threads to use during execution of parallel regions at the corresponding nested level. For example, OMP_NUM_THREADS=4,2 uses 4 threads at the first nested parallel level, and 2 at the next nested parallel level.
OMP_SCHEDULE	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for *omp for* and *omp parallel for* loops that include the runtime schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
OMP_PROC_BIND	FALSE	Specifies whether executing threads should be bound to a core during execution. Allowable values are "true" and "false".
OMP_STACKSIZE		Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	64	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

Chapter 6. Using MPI

Message Passing Interface (MPI) is an industry-standard application programming interface designed for rapid data exchange between processors in a cluster application. MPI is computer software used in computer clusters that allows the processes of a parallel application to communicate with one another.

PGI provides MPI support with PGI compilers and tools. PGI compilers provide explicit support to build MPI applications on Windows using Microsoft's implementation of MPI, MS-MPI, on Mac OS X using Open MPI, and on Linux using MPICH-1, MPICH-2, MVAPICH, and Open MPI. Of course, you may always build using an arbitrary version of MPI; to do this, use the `-I`, `-L`, or `-l` option.

PGI Workstation on Linux includes MPICH-1; PGI Workstation on Mac OS X includes Open MPI; PGI Workstation on Windows includes MS-MPI; and the PGI CDK on Linux includes MPICH-1, MPICH-2, and MVAPICH. This chapter describes how to use the MPI capabilities of PGI compilers and how to configure PGI compilers so these capabilities can be used with custom MPI installations.

The debugger and profiler are enabled to support MPI applications running locally with a limited number of processes. The *PGPROF Profile Manual* and the *PGDBG Debugger Manual* describe the MPI-enabled tools in detail:

- PGPROF graphical MPI/OpenMP/multi-thread performance profiler.
- PGDBG graphical MPI/OpenMP/multi-thread symbolic debugger.

MPI Overview

This section contains general information applicable to various MPI distributions. For specific information, refer to the distribution-specific sections later in this chapter.

MPI is a set of function calls and libraries that are used to send messages between multiple processes. These processes can be located on the same system or on a collection of distributed servers. Unlike OpenMP, the distributed nature of MPI allows it to work in almost any parallel environment.

Compiling and Linking MPI Applications

The PGI compilers provide an option, `-Mmpi=`, to make building MPI applications more convenient by adding the MPI include and library directories to the compiler's include and library search paths. The compiler

determines the location of these directories using various mechanisms described in the MPI distribution-specific sections later in this chapter.

Table 6.1 lists the sub-options supported by `-Mmpi=`.

Table 6.1. MPI Distribution Options

This MPI implementation...	Requires compiling and linking with this option...
MPICH-1	`-Mmpi=mpich1`
MPICH-2	`-Mmpi=mpich2`
MVAPICH	`-Mmpi=mvapich1`
MS-MPI	`-Mmpi=msmpi`
Open MPI	`-Mmpi` not supported. Use compiler wrappers such as mpif90.

Due to complexities in the Open MPI implementation, the `-Mmpi=openmpi` is not supported. To build using Open MPI, use the Open MPI-supplied wrappers mpicc, mpic++, mpif77, or mpif90 to compile and link. On Mac OS X these compiler wrappers are provided with PGI software. On Linux, you'll configure these compiler wrappers yourself to use PGI compilers, as described at *www.open-mpi.org*

Debugging MPI Applications

The PGI debugger, PGDBG, provides support for symbolic debugging of MPI applications. The number and location of processes that can be debugged is limited by your license. PGI Workstation licenses limit processes to a single system whereas PGI CDK licenses support general development on clusters.

For all distributions of MPI except MPICH-1, you can initiate an MPI debugging session from either the command line or from within PGDBG. For MPICH-1, debugging must be initiated at the command line. For specific information on how to initiate a debugging session for a particular version of MPI, refer to the *PGDBG Debugger Manual*.

PGDBG can display the contents of message queues for instances of MPI that have been configured to support that feature. The version of MPICH-1 provided with PGI Workstation is configured with this support. MPICH-2 or MVAPICH must be built and configured correctly to enable message queues. At this time, MS-MPI does not support displaying message queue contents.

For more information on MPI and displaying message queues, refer to the documentation for your specific distribution of MPI.

Profiling MPI Applications

The PGI performance profiler, PGPROF, provides support for profiling MPI applications. The number of processes that can be profiled is limited by your license. PGI Workstation licenses limit processes to a single system whereas PGI CDK licenses support general development on clusters PGPROF instrumentation is inserted into the program by the compiler, and after the program is executed, the PGPROF profiler can display MPI message count statistics as they relate to the source code of the application and the time spent in those portions of the application.

To create and view a performance profile of your MPI application, you must first build an instrumented version of the application using the `-Mprof=` option to specify one of the MPI distributions. The `-Mprof=` option requires that you use another profiling sub-option in conjunction with the MPI distribution sub-options, listed in Table 6.2.

Table 6.2. MPI Profiling Options

This MPI distribution...	Requires compiling and linking with this profiling option...
MPICH-1	`-Mprof=mpich1,{func|lines|time}`
MPICH-2	`-Mprof=mpich2,{func|lines|time}`
MVAPICH	`-Mprof=mvapich1,{func|lines|time}`
MS-MPI	`-Mprof=msmpi,{func|lines}`
Open MPI	Use with Open MPI compiler wrappers On Linux: `-Mprof={func|lines|time}` On Mac OS X: `-Mprof={func|lines}`

For example, you can use the following command to compile for profiling with MPICH-2:

```
% pgfortran -fast -Mprof=mpich2,func my_mpi_app.f90
```

Note

The default versions of the compiler wrappers (i.e. `mpicc` and `mpif90`) provided by some MPI distributions do not correctly support the `-Mprof` option. For best results, use the PGI compiler drivers in place of these scripts.

Once you have built an instrumented version of your MPI application, running it produces the profile data. For specific details on using PGPROF to view the profile data, refer to the *PGPROF Profiler Manual*.

Using MPICH-1 on Linux

PGI Workstation and *PGI CDK* for Linux include MPICH-1 libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs. PGI Workstation can be installed on a single system, and that system can be treated as if it is a small cluster.

Example

Example 6.1. MPI Hello World Example

The following MPI "hello world" example program uses MPICH-1.

```
% cd my_example_dir
% cp -r $PGI/linux86/13.10-0/EXAMPLES/MPI/mpihello .
% cd mpihello
% pgf77 -o mpihello mpihello.f -Mmpi=mpich1
```

```
% mpirun mpihello
Hello world! I'm node 0
```

```
% mpirun -np 4 mpihello
Hello world! I'm node 0
Hello world! I'm node 2
Hello world! I'm node 1
Hello world! I'm node 3
```

If you want to build your MPI application using the instance of MPICH-1 installed with the PGI compilers, just use the `-Mmpi=mpich1` option. Add `-g` for debugging, or use `-Mprof=mpich1` instead to instrument for MPICH-1 profiling.

To use a different instance of MPICH-1, set the `MPIDIR` environment variable before invoking the compiler. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MPICH-1 installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using MPICH-2 on Linux

PGI CDK for Linux includes MPICH-2 libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs.

If you want to build your MPI application using the instance of MPICH-2 installed with the PGI compilers, you need to append the location of `libmpl.so.1` to the `LD_LIBRARY_PATH` environment variable.

For 32-bit:

```
%setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":$PGI/linux86/2013/mpi2/mpich/lib:/$PGI
/linux86/13.10/libso
```

For 64-bit:

```
%setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":$PGI/linux86-64/2013/mpi2/mpich/lib:$PGI
/linux86-64/13.10/libso
```

You may need to put the preceding commands in `.cshrc/.bashrc` when running a program on slave nodes.

Then add the `-Mmpi=mpich2` option to the compilation and link steps; use `-Mprof=mpich2` to instrument for MPICH-2 profiling. The `-Mmpi=mpich2` option automatically sets up the include and library paths to use the MPICH-2 headers and libraries. For example, you can use the following command to compile for profiling with MPICH-2:

```
% pgfortran -fast -Mprof=mpich2,time my_mpi_app.f90
```

To use a different instance of MPICH-2, set the `MPIDIR` environment variable before invoking and linking with `-Mmpi=mpich2`. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MPICH-2 installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using MVAPICH on Linux

PGI CDK for Linux includes MVAPICH libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs.

If you want to build your MPI application using the instance of MVAPICH installed with the PGI compilers, just add the `-Mmpi=mvapich1` option to the compilation and link steps; use `-Mprof=mvapich1` to instrument for MVAPICH profiling. The `-Mmpi=mvapich1` option automatically sets up the include and library paths to use the MVAPICH headers and libraries. For example, you can use the following command to compile for profiling with MVAPICH:

```
% pgfortran -fast -Mprof=mvapich1,time my_mpi_app.f90
```

To use a different instance of MVAPICH, set the `MPIDIR` environment variable before invoking and linking with `-Mmpi=mvapich1`. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MVAPICH installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using Open MPI on Linux and Mac OS X

PGI compilers and tools support Open MPI on Linux and on Mac OS X; the instructions for using Open MPI once it is installed are the same on both operating systems. PGI ships a pre-built, pre-configured version of Open MPI on Mac OS X but not on Linux. You can build your own version of Open MPI for Linux using PGI compilers; refer to www.open-mpi.org for information on building Open MPI and configuring the Open MPI compiler wrappers for use with PGI compilers

To build an application using Open MPI, use the Open MPI compiler wrappers: mpicc, mpic++, mpif77, and mpif90. These wrappers automatically set up the compiler commands with the correct include file search paths, library directories, and link libraries. The Open MPI Project recommends use of their compiler wrappers and therefore PGI compilers do not directly support the `-Mmpi=openmpi` option.

To build an application using Open MPI for debugging, add `–g` to the compiler wrapper command line arguments

To build an application that generates MPI profile data suitable for use with PGPROF, use the Open MPI compiler wrappers with the `-Mprof=func`, `-Mprof=lines`, or `-Mprof=time` (linux only) option. If you are using these wrappers on Linux, you will first need to modify them to support `-Mprof` options as directed in the *PGPROF Profiler Manual*. PGI has pre-configured these wrappers for `-Mprof` on Mac OS X.

Using MS-MPI on Windows

PGI products on Windows include a version of Microsoft's MPI. You can compile, run, debug, and profile locally on your system using this instance of MS-MPI. You can also use this version of MS-MPI on a Microsoft HPC Server cluster.

To compile the application, use the `-Mmpi=msmpi` option. This option automatically sets up the appropriate include and library paths to use the MS-MPI headers and libraries. To compile for debugging, add `–g`.

To build an application that generates MPI profile data, use the `-Mprof=msmpi` option. This option performs MPICH-style profiling for Microsoft MPI. Using this option implies the option `-Mmpi=msmpi`. The profile data generated by running an application built with the option `-Mprof=msmpi` contains information about the number of sends and receives, as well as the number of bytes sent and received, correlated with the source location associated with the sends and receives. You must use `-Mprof=msmpi` in conjunction with either `-Mprof=func` or `-Mprof=lines`.

Using mpi Scripts

For MPICH1 and MVAPICH, if you use MPI scripts, such as `mpicc` to build with the `-fpic` or `-mcmodel=medium` options, then you must specify `-shlib` to link with the correct libraries. Here are a few examples:

For a static link to the mpi library, use this command:

```
% mpicc hello.f
```

For a dynamic link to the mpi library, use this command:

```
% mpicc hello.f -shlib
```

To compile with `-fpic`, which, by default, is a dynamic link, use this command:

```
% mpicc -fpic -shlib hello.f
```

To compile with `-mcmodel=medium`, use this command:

```
% mpicc -mcmodel=medium -shlib hello.f
```

Site-specific Customization

You can configure MPI compilers to use custom MPI installations. The section "Site-specific Customization of the Compilers," on page 13 of Chapter 1, "*Getting Started*" describes how to use the `siterc` file to customize the compiler to add certain libraries as well as include paths. This section describes how you can use the siterc file or environment variables to specify MPI installations other than the PGI-installed defaults.

- You set environment variables to change the MPI installation used by the PGI compilers for a single user or a single build.
- You change the siterc file to change the defaults for anyone using a PGI installation.

Use Alternate MPICH Installation

Important

In this example, the location of your installation is `/opt/mpi`.

To use an alternate MPICH installation, do one of the following.

- Add the following line to the `siterc` file:

  ```
  set MPIUDIR=/opt/mpi;
  ```

  OR

- Set the environment variable `MPIDIR`

  ```
  setenv MPIDIR /opt/mpi
  export MPIDIR=/opt/mpi
  ```

Once you have done this, when compiling with `-Mmpi` with any library setting, these new settings are used instead of the PGI-installed default.

Use Alternate MVAPICH Installation

Important

In this example, the location of your MVAPICH installation is `/opt/mvapich` and openfabrics is installed in `/opt/ofed`

To use an alternate MVAPICH installation, do the following.

Add the following lines to the `siterc` file:

```
set MPIVDIR=/opt/mpi;
set OFEDLIBDIR=/opt/ofed/lib;
```

Once you have done this, when compiling with `-Mmpi=mvapich1` with any library setting, these new settings are used instead of the PGI-installed default.

Use Alternate MS-MPI Installation

To use an alternate MS-MPI installation, do the following.

Important

In this example, the location of your MS-MPI installation is `C:\mymsmpi` and the MS-MPI header files are in the `Inc` directory.

Add the following lines to the `siterc` file:

```
set MSMPIINC=C:\mymsmpi\Inc
set MSMPILIB32=C:\mymsmpi\Lib\i386
set MSMPILIB64=C:\mymsmpi\Lib\amd64
```

Once you have done this, when compiling with `-Mmpi=msmpi` with any library setting, these new settings are used instead of the PGI-installed default.

Limitations

The Open Source Cluster utilities, in particular the MPICH and ScaLAPACK libraries, are provided with support necessary to build and define their proper use. However, use of these libraries on linux86-64 systems is subject to the following limitations:

- MPI libraries are limited to Messages of length < 2GB, and integer arguments are *INTEGER*4* in FORTRAN, and *int* in C.
- Integer arguments for ScaLAPACK libraries are *INTEGER*4* in FORTRAN, and *int* in C.
- Arrays passed must be < 2GB in size.

Testing and Benchmarking

The `bench` directory contains various benchmarks and tests. Copy this directory into a local working directory by issuing the following command:

```
% cp -r $PGI/linux86/13.10/EXAMPLES/MPI .
```

NAS Parallel Benchmarks

The NPB2.3 subdirectory contains version 2.3 of the NAS Parallel Benchmarks in MPI. Issue the following commands to run the BT benchmark on 4 nodes of your cluster:

```
% cd MPI/NPB2.3/BT
% make BT NPROCS=4 CLASS=W
% cd ../bin
% mpirun -np 4 bt.W.4
```

There are several other NAS parallel benchmarks available in this directory. Similar commands are used to build and run each of them. If you want to run a larger problem, try building the Class A version of BT by substituting "A" for "W" in the previous commands.

ScaLAPACK

The ScaLaPack test times execution of the 3D PBLAS (parallel BLAS) on your cluster. To run this test, execute the following commands:

```
% cd scalapack
% make
% mpirun -np 4 pdbla3tim
```

Chapter 7. Using an Accelerator

An accelerator is a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations. This chapter describes a collection of compiler directives used to specify regions of code in Fortran and C programs that can be offloaded from a *host* CPU to an attached *accelerator*.

Overview

The programming model and directives described in this chapter allow programmers to create high-level *host+accelerator* programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the PGI Fortran , C, and C++ accelerator compilers.

The method described provides a model for accelerator programming that is portable across operating systems and various host CPUs and accelerators. The directives allow a programmer to migrate applications incrementally to accelerator targets using standards-compliant Fortran or C.

This programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator region, guidance on mapping of loops onto an accelerator, and similar performance-related details.

Components

The PGI Accelerator compiler technology includes the following components:

- PGF95 auto-parallelizing accelerator-enabled Fortran 90/95 and F2003 compilers
- PGCC auto-parallelizing accelerator-enabled ANSI C99 and K&R C compiler
- NVIDIA CUDA Toolkit components
- A simple command-line tool to detect whether the system has an appropriate GPU or accelerator card

No accelerator-enabled debugger is included with this release

Availability

The PGI 13.10 Fortran & C Accelerator compilers are available only on x86 processor-based workstations and servers with an attached NVIDIA CUDA-enabled GPU or Tesla card. These compilers target all platforms that PGI supports. All examples included in this chapter are developed and presented on such a platform. For a list of supported GPUs, refer to the Accelerator Installation and Supported Platforms list in the latest PGI Release Notes.

User-directed Accelerator Programming

In user-directed accelerator programming the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The bulk of a user's program, as well as regions containing constructs that are not supported on the targeted accelerator, are executed on the host. This chapter concentrates on specification of loops and regions of code to be offloaded to an accelerator.

Features Not Covered or Implemented

This chapter does not describe features or limitations of the host programming environment as a whole. Further, it does not cover automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. While future versions of the PGI compilers may allow for automatic offloading or multiple accelerators of different types, these features are not currently supported.

Terminology

Clear and consistent terminology is important in describing any programming model. This section provides definitions of the terms required for you to effectively use this chapter and the associated programming model.

Accelerator
: a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations.

Compute intensity
: for a given loop, region, or program unit, the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.

Compute region
: a region defined by an Accelerator compute region directive. A compute region is a structured block containing loops which are compiled for the accelerator. A compute region may require device memory to be allocated and data to be copied from host to device upon region entry, and data to be copied from device to host memory and device memory deallocated upon exit. Compute regions may not contain other compute regions or data regions.

CUDA
: stands for Compute Unified Device Architecture; the CUDA environment from NVIDIA is a C-like programming environment used to explicitly control and program an NVIDIA GPU.

Data region
: a region defined by an Accelerator data region directive, or an implicit data region for a function or subroutine containing Accelerator directives. Data regions typically require device memory to be allocated

and data to be copied from host to device memory upon entry, and data to be copied from device to host memory and device memory deallocated upon exit. Data regions may contain other data regions and compute regions.

Device
: a general reference to any type of accelerator.

Device memory
: memory attached to an accelerator which is physically separate from the host memory.

Directive
: in C, a #pragma, or in Fortran, a specially formatted comment statement that is interpreted by a compiler to augment information about or specify the behavior of the program.

DMA
: Direct Memory Access, a method to move data between physically separate memories; this is typically performed by a DMA engine, separate from the host CPU, that can access the host physical memory as well as an IO device or GPU physical memory.

GPU
: a Graphics Processing Unit; one type of accelerator device.

GPGPU
: General Purpose computation on Graphics Processing Units.

Host
: the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

Loop trip count
: the number of times a particular loop executes.

OpenACC
: a parallel programming standard describing a set of compiler directives which can be applied to standard C, C++, and Fortran to specify regions of code for offloading from a host CPU to an attached accelerator.

OpenCL - Open Compute Language
: a standard C-like programming environment similar to CUDA that enables portable low-level general-purpose programming on GPUs and other accelerators.

Private data
: with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

Region
: a structured block identified by the programmer or implicitly defined by the language. Certain actions may occur when program execution reaches the start and end of a region, such as device memory allocation or data movement between the host and device memory. Loops in a compute region are targeted for execution on the accelerator.

Structured block
: in C, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

Vector operation
: a single operation or sequence of operations applied uniformly to each element of an array.

Visible device copy
: a copy of a variable, array, or subarray allocated in device memory, that is visible to the program unit being compiled.

System Requirements

To use the PGI Accelerator compiler features, you must install the NVIDIA drivers. You may download these components from the NVIDIA website at

```
www.nvidia.com/cuda
```

These are not PGI products. They are licensed and supported by NVIDIA.

Note

You must be using an operating system that is supported by both the current PGI release and by the CUDA software and drivers.

Supported Processors and GPUs

This PGI Accelerator compiler release supports all AMD64 and Intel 64 host processors. Use the `-tp <target>` flag as documented in the release to specify the target processor.

Use the `-ta=nvidia` flag to enable the accelerator directives and target the NVIDIA GPU. You can then use the generated code on any system with CUDA installed that has a CUDA-enabled GeForce, Quadro, or Tesla card.

Use the `-acc` flag to enable the OpenACC directives and the OpenACC runtime.

For more information on these flags as they relate to accelerator technology, refer to "Applicable Command Line Options," on page 96.

For a complete list of supported CUDA GPUs, refer to the NVIDIA website at:

```
www.nvidia.com/object/cuda_learn_products.html
```

You can detect whether the system has CUDA properly installed and has an attached GPU by running the **pgaccelinfo** command, which is delivered as part of the PGI Accelerator compilers software package.

Installation and Licensing

Note

The PGI Accelerator compilers have a different license key than the -x64 only version of the PGI Workstation, PGI Server, or PGI CDK products.

Required Files

Note

If you are installing on Windows, the required files are built for you.

The default NVIDIA Compute Capability for generated code is now both cc1.x and cc2.0, enabling code generation for both compute capabilities. cc1.x is the lowest compute capability that includes all the features used in the generated code. For instance, if double precision is used, then the lowest compute capability is 1.3.

You can change the default to one or more of the supported compute capabilities by adding a line similar to the following one to the `sitenvrc` file. This example sets the compute capability to enable code generation for all of the supported compute capabilities. Notice that the compute capabilities are separated by a space.

```
set COMPUTECAP=10 11 12 13 20 30;
```

Place the `sitenvrc` file in the following directory, where $PGI is the PGI installation directory, which is typically `/opt/pgi` or `/usr/pgi`.

```
$PGI/linux86-64/13.10/bin/
```

Command Line Flag

After creating the `sitenvrc` file and acquiring the PGI Accelerator compilers license key, you can use the option `-ta=nvidia` with the `pgfortran` or `pgcc` commands.

For more information on the `-ta` flag and the suboptions that relate to the target accelerators, refer to "Applicable Command Line Options," on page 96.

The compiler automatically invokes the necessary CUDA software tools to create the kernel code and embeds the kernels in the Linux object file.

Note

To access the accelerator libraries, you must link an accelerator program with the `-ta` flag as well.

Execution Model

The execution model targeted by the PGI Accelerator compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The accelerator device executes kernels, which may be as simple as a tightly-nested loop, or as complex as a subroutine, depending on the accelerator hardware.

Host Functions

Even in accelerator-targeted regions, the host must orchestrate the execution; it

- allocates memory on the accelerator device

- initiates data transfer
- sends the kernel code to the accelerator
- passes kernel arguments
- queues the kernel
- waits for completion
- transfers results back to the host
- deallocates memory

Note

In most cases, the host can queue a sequence of kernels to be executed on the device, one after the other.

Levels of Parallelism

Most current GPUs support two levels of parallelism:

- an outer *doall* (fully parallel) loop level
- an inner *synchronous* (SIMD or vector) loop level

Each level can be multidimensional with 2 or 3 dimensions, but the domain must be strictly rectangular. The *synchronous* level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the *doall* level.

The execution model on the device side exposes these two levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization across iterations. All fully parallel loops can be scheduled for either doall or synchronous parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

Memory Model

The most significant difference between a *host-only* program and a *host+accelerator* program is that the memory on the accelerator can be completely separate from host memory, which is the case on most current GPUs. For example:

- The host cannot read or write accelerator memory by reference because it is not mapped into the virtual memory space of the host.
- All data movement between host memory and accelerator memory must be performed by the host through runtime library calls that explicitly move data between the separate memories.
- It is not valid to assume the accelerator can read or write host memory, though this may be supported by accelerators in the future.

Separate Host and Accelerator Memory Considerations

The programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and accelerator memory determines the compute intensity required to effectively accelerate a given region of code.
- Limited size of accelerator memory may prohibit offloading of regions of code that operate on very large amounts of data.

Accelerator Memory

On the accelerator side, current GPUs implement a weak memory model. In particular, they do not support memory coherence between threads unless those threads are parallel only at the synchronous level and the memory operations are separated by an explicit barrier. Otherwise, if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. While the results of running such a program might be inconsistent, it is not accurate to say that the results are incorrect. By definition, such programs are defined as being in error. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator region that produces inconsistent numerical results.

Cache Management

Some current GPUs have a software-managed cache, some have hardware-managed caches, and most have hardware caches that can be used only in certain situations and are limited to read-only data. In low-level programming models such as CUDA or OpenCL, it is up to the programmer to manage these caches. However, in the PGI Accelerator programming model, the compiler manages these caches using hints from the programmer in the form of directives.

Running an Accelerator Program

Running a program that has accelerator directives and was compiled and linked with the `-ta=nvidia` flag is the same as running the program compiled without the `-ta=nvidia` flag.

- The program looks for and dynamically loads the CUDA libraries. If the libraries are not available, or if they are in a different directory than they were when the program was compiled, you may need to append the CUDA library directory to your LD_LIBRARY_PATH environment variable on Linux or to the PATH environment variable on Windows.
- On Linux, if you have no server running on your NVIDIA GPU, when your program reaches its first accelerator region, there may be a 0.5 to 1.5 second pause to warm up the GPU from a power-off condition. You can avoid this delay by running the `pgcudainit` program in the background, which keeps the GPU powered on.
- If you run an accelerated program on a system without a CUDA-enabled NVIDIA GPU, or without the CUDA software installed in a directory where the runtime library can find it, the program fails at runtime with an error message.

- If you set the environment variable ACC_NOTIFY to a nonzero integer value, the runtime library prints a line to standard error every time it launches a kernel.

Accelerator Directives

This section provides an overview of the Fortran and C directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data.

Enable Accelerator Directives

PGI Accelerator compilers enable accelerator directives with the -ta command line option. For more information on this option as it relates to the Accelerator, refer to "Applicable Command Line Options," on page 96.

Note

The syntax used to define directives allows compilers to ignore accelerator directives if support is disabled or not provided.

_ACCEL macro

The _ACCEL macro name is defined to have a value yyyymm where yyyy is the year and mm is the month designation of the version of the Accelerator directives supported by the implementation. For example, the version for May, 2009 is 200905. This macro must be defined by a compiler when accelerator directives are enabled.

Format

The specific format of the directive depends on the language and the format or form of the source.

Directives include a name and clauses, and the format of the directive depends on the type:

- C directives, described in "C Directives"
- Free-form Fortran directives, described in "Free-Form Fortran Directives"
- Fixed-form Fortran directives, described in "Fixed-Form Fortran Directives"

Note

This document uses free form for all PGI Accelerator compiler Fortran directive examples.

Rules

The following rules apply to all PGI Accelerator compiler directives:

- Only one directive-name can be specified per directive.
- The order in which clauses appear is not significant.
- Clauses may be repeated unless otherwise specified.

- For clauses that have a *list* argument, a list is a comma-separated list of variable names, array names, or, in some cases, subarrays with subscript ranges.

C Directives

In C, PGI Accelerator compiler directives are specified using `#pragma`

Syntax

The syntax of a PGI Accelerator compiler directive is:

```
#pragma acc directive-name [clause [,clause]…] new-line
```

Rules

In addition to the general directive rules, the following rules apply to PGI Accelerator compiler C directives:

- Each directive starts with `#pragma acc`.
- The remainder of the directive follows the C conventions for pragmas.
- White space may be used before and after the #; white space may be required to separate words in a directive.
- Preprocessing tokens following the `#pragma acc` are subject to macro replacement.
- C directives are case sensitive.
- An Accelerator directive applies to the immediately following structured block or loop.

Free-Form Fortran Directives

PGI Accelerator compiler Fortran directives can be either Free-Form or Fixed-Form directives. Free-Form Accelerator directives are specified with the `!$acc` mechanism.

Syntax

The syntax of directives in free-form source files is:

```
!$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to PGI Accelerator compiler Free-Form Fortran directives:

- The comment prefix (!) may appear in any column, but may only be preceded by white space (spaces and tabs).
- The sentinel (!$acc) must appear as a single word, with no intervening white space.
- Line length, white space, and continuation rules apply to the directive line.

- Initial directive lines must have a space after the sentinel.
- Continued directive lines must have an ampersand (&) as the last nonblank character on the line, prior to any comment placed in the directive.
- Comments may appear on the same line as the directive, starting with an exclamation point and extending to the end of the line.
- If the first nonblank character after the sentinel is an exclamation point, the line is ignored.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Fixed-Form Fortran Directives

Fixed-Form Accelerator directives are specified using one of three formats.

Syntax

The syntax of directives in fixed-form source files is one these three formats:

```
!$acc directive-name [clause [,clause]…]
c$acc directive-name [clause [,clause]…]
*$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to Accelerator Fixed-Form Fortran directives:

- The sentinel (!acc, cacc, or *$acc) must occupy columns 1-5.
- Fixed form line length, white space, continuation, and column rules apply to the directive line.
- Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or zero in column 6.
- Comments may appear on the same line as a directive, starting with an exclamation point on or after column 7 and continuing to the end of the line.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Accelerator Directive Summary

PGI currently supports these types of accelerator directives:

Accelerator Compute Region Directive
Accelerator Loop Mapping Directive
Combined Directive
Accelerator Declarative Data Directive
Accelerator Update Directive

Table 7.1 lists and briefly describes each of the accelerator directives that PGI currently supports. For a complete description of each directive, refer to "PGI Accelerator Directives" in the PGI Compiler Reference Manual.

Table 7.1. PGI Accelerator Directive Summary Table

This directive...	Accepts these clauses...	Has this functionality...
Accelerator Compute Region Directive	if(condition) copy (*list*) copyin(*list*) copyout(*list*) local(*list*) updatein(*list*) updateout(*list*)	Defines the region of the program that should be compiled for execution on the accelerator device.

C Syntax
`#pragma acc region [clause [, clause]…] new-line`
`structured block`
Fortran Syntax
`!$acc region [clause [, clause]…]`
`*structured block*`
`!$acc end region`		

Accelerator Data Region Directive	copy (*list*) copyin(*list*) copyout(*list*) local(*list*) mirror(*list*) updatein(*list*) updateout(*list*)	Defines data, typically arrays, that should be allocated in the device memory for the duration of the data region, whether data should be copied from the host to the device memory upon region entry, and copied from the device to host memory upon region exit.

C Syntax
`#pragma acc data region [clause [, clause]…] new-line`
`structured block`
Fortran Syntax
`!$acc data region [clause [, clause]…]`
`*structured block*`
`!$acc end data region`		

This directive...	Accepts these clauses...	Has this functionality...
Accelerator Loop Mapping Directive	cache(*list*) host [(*width*)] independent kernel parallel [(*width*)] private(*list*) seq [(*width*)] unroll [(*width*)] vector [(*width*)]	Describes what type of parallelism to use to execute the loop and declare loop-private variables and arrays. Applies to a loop which must appear on the following line.

C Syntax
`#pragma acc for [clause [,clause]...]new-line`
`for loop`
Fortran Syntax
`!$acc do [clause [,clause]…]`
`do loop`		

Combined Directive	Any clause that is allowed on a region directive or a loop directive is allowed on a combined directive.	Is a shortcut for specifying a loop directive nested immediately inside an accelerator compute region directive. The meaning is identical to explicitly specifying a region construct containing a loop directive.

C Syntax
`#pragma acc region for [clause [, clause]...] new-line`
`for loop`
Fortran Syntax
`!$acc region do [clause [, clause]...]`
`do loop`		

Accelerator Declarative Data Directive	copy (*list*) copyin(*list*) copyout(*list*) local(*list*) mirror (*list*) reflected(*list*)	Specifies that an array or arrays are to be allocated in the device memory for the duration of the implicit data region of a function, subroutine, or program. Specifies whether the data values are to be transferred from the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region. Creates a visible device copy of the variable or array.

C Syntax
`#pragma acc declclause [,declclause]...new-line`
Fortran Syntax
`!$acc declclause [,declclause]...`		

This directive...	Accepts these clauses...	Has this functionality...
Accelerator Update Directive	host (*list*) device(*list*)	Used within an explicit or implicit data region to update all or part of a host memory array with values from the corresponding array in device memory, or to update all or part of a device memory array with values from the corresponding array in host memory.

C Syntax

```
#pragma acc update updateclause [,updateclause]...new-line
```

Fortran Syntax

```
   !$acc update updateclause [,updateclause]...
```

Accelerator Directive Clauses

Table 7.2provides an alphabetical listing and brief description of each clause that is applicable for the various Accelerator directives. The table also indicates for which directives the clause is applicable.

For more information on the restrictions and use of each clause, refer to the "PGI Accelerators Directive Clauses" section in the PGI Compiler Reference Manual.

Table 7.2. Directive Clauses Summary

Use this clause...	In these directives...	To do this...
cache (list)	Accelerator Loop Mapping	Provides a hint to the compiler to try to move the variables, arrays, or subarrays in the list to the highest level of the memory hierarchy.
copy (list)	Accelerator Data Region Declarative Data	Declares that the variables, arrays, or subarrays in the list have values in the host memory that need to be copied to the accelerator memory, and are assigned values on the accelerator that need to be copied back to the host.
copyin (list)	Accelerator Data Region Declarative Data	Declares that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory.
copyout (list)	Accelerator Data Region Declarative Data	Declares that the variables, arrays, or subarrays in the list are assigned or contain values in the accelerator memory that need to be copied back to the host memory at the end of the accelerator region.
device (list)	Update	Copies the variables, arrays, or subarrays in the list argument from host memory to the visible device copy of the variables, arrays, or subarrays in device memory. Copy occurs before beginning execution of the compute or data region.

Use this clause...	In these directives...	To do this...
host (list)	Update	Copies the visible device copies of the variables, arrays, or subarrays in the *list* argument to the associated host memory locations. The copy occurs after completion of the compute or data region.
host [(width)]	Accelerator Loop Mapping	Tells the compiler to execute the loop sequentially on the host processor.
if (condition)	Accelerator Compute Data Region	When present, tells the compiler to generate two copies of the region - one for the accelerator, one for the host - and to generate code to decide which copy to execute.
independent	Accelerator Loop Mapping	Tells the compiler that the iterations of this loop are data-independent of each other, thus allowing the compiler to generate code to examine the iterations in parallel, without synchronization.
kernel	Accelerator Loop Mapping	Tells the compiler that the body of this loop is to be the body of the computational kernel. Any loops contained within the kernel loop are executed sequentially on the accelerator.
local (list)	Accelerator Data Region Declarative Data	Declares that the variables, arrays or subarrays in the *list* need to be allocated in the accelerator memory, but the values in the host memory are not needed on the accelerator, and the values computed and assigned on the accelerator are not needed on the host.
mirror (list)	Accelerator Data Region Declarative Data	Declares that the arrays in the *list* need to mirror the allocation state of the host array within the region. Valid only in Fortran on Accelerator data region directive.
parallel [(width)]	Accelerator Loop Mapping	Tells the compiler to execute this loop in parallel mode on the accelerator. There may be a target-specific limit on the number of iterations in a parallel loop or on the number of parallel loops allowed in a given kernel
private (list)	Accelerator Loop Mapping	Declares that the variables, arrays, or subarrays in the *list* argument need to be allocated in the accelerator memory with one copy for each iteration of the loop.
reflected (list)	Declarative Data	Declares that the actual argument arrays that are bound to the dummy argument arrays in the *list* need to have a visible copy at the call site.
seq [(width)]	Accelerator Loop Mapping	Tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule.

Use this clause...	In these directives...	To do this...
unroll [(width)]	Accelerator Loop Mapping	Tells the compiler to unroll *width* iterations for sequential execution on the accelerator. The *width* argument must be a compile time positive constant integer.
updatein (list)	Accelerator Data Region	Copies the variables, arrays, or subarrays in the *list* argument from host memory to the visible device copy of the variables, arrays, or subarrays in device memory, before beginning execution of the compute or data region.
updateout (list)	Accelerator Data Region	Copies the visible device copies of the variables, arrays, or subarrays in the *list* argument to the associated host memory locations, after completion of the compute or data region.
vector [(width)]	Accelerator Loop Mapping	Tells the compiler to execute this loop in vector mode on the accelerator.

PGI Accelerator Compilers Runtime Libraries

This section provides an overview of the user-callable functions and library routines that are available for use by programmers to query the accelerator features and to control behavior of accelerator-enabled programs at runtime.

Note

In Fortran, none of the PGI Accelerator compilers runtime library routines may be called from a PURE or ELEMENTAL procedure.

Runtime Library Definitions

There are separate runtime library files for C and for Fortran.

C Runtime Library Files

In C, prototypes for the runtime library routines are available in a header file named `accel.h`. All the library routines are `extern` functions with "C" linkage. This file defines:

- The prototypes of all routines in this section.
- Any data types used in those prototypes, including an enumeration type to describe types of accelerators.

Fortran Runtime Library Files

In Fortran, interface declarations are provided in a Fortran include file named `accel_lib.h` and in a Fortran module named `accel_lib`. These files define:

- Interfaces for all routines in this section.
- Integer parameters to define integer kinds for arguments to those routines.
- Integer parameters to describe types of accelerators.
- The integer parameter `accel_version` with a value `yyyymm` where `yyyy` and `mm` are the year and month designations of the version of the Accelerator programming model supported. This value matches the value of the preprocessor variable `_ACCEL`.

Runtime Library Routines

Table 7.3 lists and briefly describes the supported PGI Accelerator compilers runtime library routines. For a complete description of these routines, refer to the PGI Accelerator Runtime Routines chapter in the PGI Compiler Reference Manual.

Table 7.3. Accelerator Runtime Library Routines

This Runtime Library Routine...	Does this...
acc_allocs	Returns the number of arrays allocated in data or compute regions.
acc_bytesalloc	Returns the total bytes allocated by data or compute regions.
acc_bytesin	Returns the total bytes copied in to the accelerator by data or compute regions.
acc_bytesout	Returns the total bytes copied out from the accelerator by data or compute regions.
acc_copyins	Returns the number of arrays copied in to the accelerator by data or compute regions.
acc_copyouts	Returns the number of arrays copied out from the accelerator by data or compute regions.
acc_disable_time	Tells the runtime to stop profiling accelerator regions and kernels.
acc_enable_time	Tells the runtime to start profiling accelerator regions and kernels, if it is not already doing so.
acc_exec_time	Returns the number of microseconds spent on the accelerator executing kernels.
acc_free	Frees memory allocated on the attached accelerator. [C only]
acc_frees	Returns the number of arrays freed or deallocated in data or compute regions.
acc_get_device	Returns the type of accelerator device used to run the next accelerator region, if one is selected.
acc_get_device_num	Returns the number of the device being used to execute an accelerator region.
acc_get_free_memory	Returns the total available free memory on the attached accelerator device.
acc_get_memory	Returns the total memory on the attached accelerator device.

This Runtime Library Routine...	Does this...
acc_get_num_devices	Returns the number of accelerator devices of the given type attached to the host.
acc_init	Connects to and initializes the accelerator device and allocates control structures in the accelerator library.
acc_kernels	Returns the number of accelerator kernels launched since the start of the program.
acc_malloc	Allocates memory on the attached accelerator. [C only]
acc_on_device	Tells the program whether it is executing on a particular device.
acc_regions	Returns the number of accelerator regions entered since the start of the program.
acc_set_device	Tells the runtime which type of device to use when executing an accelerator compute region.
acc_set_device_num	Tells the runtime which device of the given type to use among those that are attached.
acc_shutdown	Tells the runtime to shutdown the connection to the given accelerator device, and free up any runtime resources.
acc_total_time	Returns the number of microseconds spent in accelerator compute regions and in moving data for accelerator data regions.

Environment Variables

PGI supports environment variables that modify the behavior of accelerator regions. This section defines the user-setable environment variables used to control behavior of accelerator-enabled programs at execution. These environment variables must comply with these rules:

- The names of the environment variables must be upper case.
- The values assigned environment variables are case insensitive and may have leading and trailing white space.
- The behavior is implementation-defined if the values of the environment variables change after the program has started, even if the program itself modifies the values.

Table 7.4 lists and briefly describes the Accelerator environment variables that PGI supports.

Table 7.4. Accelerator Environment Variables

This environment variable...	Does this...
ACC_DEVICE	Controls which accelerator device to use when executing accelerator regions, if the program has been compiled to use more than one different type of device. The value of this environment variable is implementation-defined, and currently may be the string NVIDIA or HOST.

This environment variable...	Does this...
ACC_DEVICE_NUM	Controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.
ACC_NOTIFY	When set to a non-negative integer, indicates to print a message to standard output when a kernel is executed on an accelerator.

Applicable Command Line Options

The following command line options are applicable specifically when working with accelerators.

`-ta`
Use this option to enable recognition of the **!$ACC** directives in Fortran, and **#pragma acc** directives in C.

`-tp`
Use this option to specify the target host processor architecture.

`-Minfo` or `-Minfo=accel`
Use this option to see messages about the success or failure of the compiler in translating the accelerator region into GPU kernels.

The `-ta` flag has the following accelerator-related suboptions:

nvidia
Select NVIDIA accelerator target. This option has a number of suboptions:

cc10, cc11, cc12, cc13, cc20, cc30, cc35	Generate code for compute capability 1.0, 1.1, 1.2, 1.3, 2.0, 3.0, or 3.5 respectively; multiple selections are valid.
cuda5.0 or 5.0	Specify the CUDA 5.0 version of the toolkit. This is the default.
cuda5.5 or 5.5	Specify the CUDA 5.5 version of the toolkit.
fastmath	Use routines from the fast math library.
fermi	Generate code for Fermi Architecture equivalent to NVIDIA compute capability 2.x.
[no]flushz	control flush-to-zero mode for floating point computations in the GPU code generated for PGI Accelerator model compute regions.
keep	Keep the kernel files.
kepler	Generate code for Kepler Architecture equivalent to NVIDIA compute capability 3.x.
maxregcount:n	Specify the maximum number of registers to use on the GPU. Leaving this blank indicates no limit.
nofma	Do not generate fused multiply-add instructions.
noL1	Prevent the use of L1 hardware data cache to cache global variables.

tesla	Generate code for Tesla Architecture equivalent to NVIDIA compute capability 1.x.
time	Link in a limited-profiling library, as described in "Profiling Accelerator Kernels," on page 99.

host
: Select NO accelerator target. Generate PGI Unified Binary Code, as described in "PGI Unified Binary for Accelerators," on page 97.

The compiler automatically invokes the necessary CUDA software tools to create the kernel code and embeds the kernels in the object file.

Note

To access accelerator libraries, you must *link* an accelerator program with the `-ta` flag.

PGI Unified Binary for Accelerators

Note

The information and capabilities described in this section are only supported for 64-bit systems.

PGI compilers support the PGI Unified Binary feature to generate executables with functions optimized for different host processors, all packed into a single binary. This release extends the PGI Unified Binary technology for accelerators. Specifically, you can generate a single binary that includes two versions of functions:

- one is optimized for the accelerator
- one runs on the host processor when the accelerator is not available or when you want to compare host to accelerator execution.

To enable this feature, use the extended `-ta` flag:

```
-ta=nvidia,host
```

This flag tells the compiler to generate two versions of functions that have valid accelerator regions.

- A compiled version that targets the accelerator.
- A compiled version that ignores the accelerator directives and targets the host processor.

If you use the `-Minfo` flag, you get messages similar to the following:

```
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=host
     18, Generated an alternate loop for the inner loop
         Generated vector sse code for inner loop
```

```
          Generated 1 prefetch instructions for this loop
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=nvidia
     15, Generating copy(b(:,2:90))
         Generating copyin(a(:,2:90))
     16, Loop is parallelizable
     18, Loop is parallelizable
         Parallelization requires privatization of array t(2:90)
         Accelerator kernel generated
         16, !$acc do parallel
         18, !$acc do parallel, vector(256)
             Using register for t
```

The PGI Unified Binary message shows that two versions of the subprogram `s1` were generated:

- one for no accelerator (`-ta=host`)
- one for the NVIDIA GPU (`-ta=nvidia`)

At run time, the program tries to load the NVIDIA CUDA dynamic libraries and test for the presence of a GPU. If the libraries are not available or no GPU is found, the program runs the host version.

You can also set an environment variable to tell the program to run on the NVIDIA GPU. To do this, set ACC_DEVICE to the value `NVIDIA` or `nvidia`. Any other value of the environment variable causes the program to use the host version.

Note

The only supported `-ta` targets for this release are `nvidia` and `host`.

Multiple Processor Targets

With 64-bit processors, you can use the `-tp` flag with multiple processor targets along with the `-ta` flag. You see the following behavior:

- If you specify one `-tp` value and one `-ta` value:

  You see one version of each subprogram generated for that specific target processor and target accelerator.

- If you specify one `-tp` value and multiple `-ta` values:

  The compiler generates two versions of subprograms that contain accelerator regions for the specified target processor and each target accelerator.

- If you specify multiple `-tp` values and one `-ta` value:

  If 2 or more `-tp` values are given, the compiler generates up to that many versions of each subprogram, for each target processor, and each version also targets the selected accelerator.

- If you specify multiple `-tp` values and multiple `-ta` values:

  With 'N' `-tp` values and two `-ta` values, the compiler generates up to N+1 versions of the subprogram. It first generates up to N versions, for each `-tp` value, ignoring the accelerator regions, which is equivalent to using `-ta=host`. It then generates one additional version with the accelerator target.

Profiling Accelerator Kernels

This release supports the command line option:

```
-ta=nvidia,time
```

The `time` suboption links in a timer library, which collects and prints out simple timing information about the accelerator regions and generated kernels.

Example 7.1. Accelerator Kernel Timing Data

```
bb04.f90
  s1
   15: region entered 1 times
  time(us): total=1490738
                init=1489138 region=1600
                kernels=155 data=1445
  w/o init: total=1600 max=1600
                min=1600 avg=1600
   18: kernel launched 1 times
  time(us): total=155 max=155 min=155 avg=155
```

In this example, a number of things are occurring:

- For each accelerator region, the file name `bb04.f90` and subroutine or function name `s1` is printed, with the line number of the accelerator region, which in the example is `15`.
- The library counts how many times the region is entered (`1` in the example) and the microseconds spent in the region (in this example `1490738`), which is split into initialization time (in this example `1489138`) and execution time (in this example `1600`).
- The execution time is then divided into kernel execution time and data transfer time between the host and GPU.
- For each kernel, the line number is given, (`18` in the example), along with a count of kernel launches, and the total, maximum, minimum, and average time spent in the kernel, all of which are 155 in this example.

Related Accelerator Programming Tools

PGPROF pgcollect

The PGI profiler, PGPROF, has an **Accelerator tab** that displays profiling information provided by the accelerator. This information is available in the file `pgprof.out` and is collected by using **pgcollect** on an executable binary compiled for an accelerator target. For more information on **pgcollect**, refer to the "pgcollect Reference" chapter of the PGPROF Profiler Manual.

NVIDIA CUDA Profile

You can use the NVIDIA CUDA Profiler with PGI-generated code for the NVIDIA GPUs. You may download the CUDA Profiler from the same website as the CUDA software:

```
www.nvidia.com/cuda
```

Documentation and support is provided by NVIDIA.

TAU - Tuning and Analysis Utility

You can use the TAU (Tuning and Analysis Utility), version 2.18.1+, with PGI-generated accelerator code. TAU instruments code at the function or loop level, and version 2.18.1 is enhanced with support to track performance in accelerator regions. TAU software and documentation is available at this website:

```
http://tau.uoregon.edu
```

Supported Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

Intrinsics make the use of processor-specific enhancements easier because they provide a language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

This section contains an overview of the Fortran and C intrinsics that the accelerator supports.

Supported Fortran Intrinsics Summary Table

Table 7.5 is an alphabetical summary of the supported Fortran intrinsics that the accelerator supports. These functions are specific to Fortran 90/95 unless otherwise specified.

Note

For complete descriptions of these intrinsics, refer to the Chapter 6, "Fortran Intrinsics" of the PGI Fortran Reference.

In most cases PGI provides support for all the data types for which the intrinsic is valid. When support is available for only certain data types, the middle column of the table specifies which ones, using the following codes:

I for integer	S for single precision real	C for single precision complex
	D for double precision real	Z for double precision complex

Table 7.5. Supported Fortran Intrinsics

This intrinsic		Returns this value ...
ABS	I,S,D	absolute value of the supplied argument.
ACOS		arccosine of the specified value.
AINT		truncation of the supplied value to a whole number.
ANINT		nearest whole number to the supplied argument.
ASIN		arcsine of the specified value.
ATAN		arctangent of the specified value.
ATAN2		arctangent of the specified value.

This intrinsic		Returns this value ...
COS	S,D	cosine of the specified value.
COSH		hyperbolic cosine of the specified value.
DBLE	S,D	conversion of the value to double precision real.
DPROD		double precision real product.
EXP	S,D	exponential value of the argument.
IAND		result of a bit-by-bit logical AND on the arguments.
IEOR		result of a bit-by-bit logical exclusive OR on the arguments.
INT	I,S,D	conversion of the value to integer type.
IOR		result of a bit-by-bit logical OR on the arguments.
LOG	S,D	natural logarithm of the specified value.
LOG10		base-10 logarithm of the specified value.
MAX		maximum value of the supplied arguments.
MIN		minimum value of the supplied arguments.
MOD	I	remainder of the division.
NINT		nearest integer to the real argument.
NOT		result of a bit-by-bit logical complement on the argument.
REAL	I,S,D	conversion of the argument to real.
SIGN		absolute value of A times the sign of B.
SIN	S,D	value of the sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D	square root of the argument.
TAN		tangent of the specified value.
TANH		hyperbolic tangent of the specified value.

Supported C Intrinsics Summary Table

This section contains two alphabetical summaries - one for double functions and a second for float functions. These lists contain only those C intrinsics that the accelerator supports.

Table 7.6. Supported C Intrinsic Double Functions

This intrinsic	Returns this value ...
acos	arccosine of the specified value.
asin	arcsine of the specified value.
atan	arctangent of the specified value.
atan2	arctangent of y/x, where y is the first argument, x the second.
cos	cosine of the specified value.

This intrinsic	Returns this value ...
cosh	hyperbolic cosine of the specified value.
exp	exponential value of the argument.
fabs	absolute value of the argument.
fmax	maximum value of the two supplied arguments
fmin	minimum value of the two supplied arguments
log	natural logarithm of the specified value.
log10	base-10 logarithm of the specified value.
pow	value of the first argument raised to the power of the second argument.
sin	value of the sine of the argument.
sinh	hyperbolic sine of the argument.
sqrt	square root of the argument.
tan	tangent of the specified value.
tanh	hyperbolic tangent of the specified value.

Table 7.7. Supported C Intrinsic Float Functions

This intrinsic	Returns this value ...
acosf	arccosine of the specified value.
asinf	arcsine of the specified value.
atanf	arctangent of the specified value.
atan2f	arctangent of y/x, where y is the first argument, x the second.
cosf	cosine of the specified value.
coshf	hyperbolic cosine of the specified value.
expf	exponential value of the floating-point argument.
fabsf	absolute value of the floating-point argument.
logf	natural logarithm of the specified value.
log10f	base-10 logarithm of the specified value.
powf	value of the first argument raised to the power of the second argument.
sinf	value of the sine of the argument.
sinhf	hyperbolic sine of the argument.
sqrtf	square root of the argument.
tanf	tangent of the specified value.
tanhf	hyperbolic tangent of the specified value.

References related to Accelerators

- ISO/IEC 1539-1:1997, Information Technology - Programming Languages - Fortran, Geneva, 1997 (Fortran 95).
- American National Standard Programming Language C, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, Information Technology - Programming Languages - C, Geneva, 1999 (C99).
- PGDBG Dubugger Manual, The Portland Group, Release 13.8, August, 2013. Available online at http://www.pgroup.com/doc/pgdbg.pdf.
- PGPROF Profiler Manual, The Portland Group, Release 13.8, August, 2013. Available online at http://www.pgroup.com/doc/pgprof.pdf.
- PGI Fortran Reference, The Portland Group, Release 13.8, August, 2013. Available online at http://www.pgroup.com/doc/pgifortref.pdf

Chapter 8. Eclipse

This document explains how to install and use the PGI plugin for Eclipse CDT (C/C++ development tool). PGI Eclipse integration is only available on Linux.

Install Eclipse CDT

To install Eclipse plugins for the PGI C and C++ compilers:

1. Before you install, check your CDT version.
    1. Go to Help -> About Eclipse
    2. Click the Eclipse CDT button.

       You might need to hover the mouse pointer on the button to see the hint.
    3. Select Eclipse C/C++ Development Tools.

       The first number in the feature version specifies which plugin version is selected.
2. Go to Help -> Install New software.
3. Click the Add button to add a new software repository.
4. In the Add Repository dialog box:
    1. Click Local.
    2. Select your PGI installation directory, such as `/opt/pgi`.
    3. Browse inside `2013/eclipse` and select the directory matching your CDT version.
    4. Click OK.

       The Add Repository dialog should show the path to the local directory containing the plugin for your CDT version. For example, if PGI compilers are installed in /opt/pgi, then the CDT 7 plugin is located in `/opt/pgi/<os-version>/2013/eclipse/cdt7`; the CDT 8 plugin is in `/opt/pgi/<os-version>/2013/eclipse/cdt8`, and so on.
    5. Click OK in the Add Repository dialog.

The install form now shows “The Portland Group C/C++ Compiler Plugin” as an option to install.

5. Check the box next to The Portland Group option and select Next to get to the Install Details view.
6. Click Next again.
7. Review and accept the End-User License agreement.
8. Click Finish.

You are prompted to restart. Select Restart to complete installation of the plugin.

Use Eclipse CDT

To use Eclipse plugins for the PGI C and C++ compilers, the directory containing PGI compilers and tools should be included in your PATH *prior* to starting Eclipse IDE. For details on how to include this directory in your PATH environment variable, refer to Chapter 11, “ *Using Environment Variables*”, and specifically to “PATH,” on page 142.

PGI plugins follow the same rules for creating, building, and running a project as any other compiler supported by Eclipse. For more information, refer to Eclipse documentation and tutorials at: *http://www.eclipse.org/documentation/*.

Chapter 9. Using Directives and Pragmas

It is often useful to be able to alter the effects of certain command line options or default behavior of the compiler. Fortran directives and C/C++ pragmas provide pragmatic information that control the actions of the compiler in a particular portion of a program without affecting the program as a whole. That is, while a command line option affects the entire source file that is being compiled, directives and pragmas apply, or disable, the effects of a command line option to selected subprograms or to selected loops in the source file, for example, to optimize a specific area of code. Use directives and pragmas to tune selected routines or loops.

PGI Proprietary Fortran Directives

PGI Fortran compilers support proprietary directives that may have any of the following forms:

```
!pgi$g directive
!pgi$r directive
!pgi$l directive
!pgi$ directive
```

Note

If the input is in fixed format, the comment character must begin in column 1 and either * or C is allowed in place of !.

The scope indicator controls the scope of the directive. This indicator occurs after the $. Some directives ignore the scope indicator.

The valid scopes, shown in the previous forms of the directive, are these:

g
: (global) indicates the directive applies to the end of the source file.

r
: (routine) indicates the directive applies to the next subprogram.

l
: (loop) indicates the directive applies to the next loop, but not to any loop contained within the loop body. Loop-scoped directives are only applied to `DO` loops.

blank
: indicates that the default scope for the directive is applied.

The body of the directive may immediately follow the scope indicator. Alternatively, any number of blanks may precede the name of the directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equal sign.

The directive name, including the directive prefix, may contain upper or lower case letters, and the case is not significant. Case is significant for any variable names that appear in the body of the directive if the command line option `–Mupcase` is selected. For compatibility with other vendors' directives, the prefix `cpgi$` may be substituted with `cdir$` or `cvd$`.

PGI Proprietary C and C++ Pragmas

Pragmas may be supplied in a C/C++ source file to provide information to the compiler. Many pragmas have a corresponding command-line option. Pragmas may also toggle an option, selectively enabling and disabling the option.

The general syntax of a pragma is:

```
#pragma [ scope ] pragma-body
```

The optional scope field is an indicator for the scope of the pragma; some pragmas ignore the scope indicator.

The valid scopes are:

global
: indicates the pragma applies to the entire source file.

routine
: indicates the pragma applies to the next function.

loop
: indicates the pragma applies to the next loop (but not to any loop contained within the loop body). Loop-scoped pragmas are only applied to for and while loops.

If a scope indicator is not present, the default scope, if any, is applied. Whitespace must appear after the pragma keyword and between the scope indicator and the body of the pragma. Whitespace may also surround any special characters, such as a comma or an equal sign. Case is significant for the names of the pragmas and any variable names that appear in the body of the pragma.

PGI Proprietary Optimization Directive and Pragma Summary

The following table summarizes the supported Fortran directives and C/C++ pragmas. The following terms are useful in understanding the table.

- Functionality is a brief summary of the way to use the directive or pragma. For a complete description, refer to the "Directives and Pragmas Reference" chapter of the PGI Compiler Reference Manual.
- Many of the directives and pragmas can be preceded by `NO`. The default entry indicates the default for the directive or pragma. N/A appears if a default does not apply.

- The scope entry indicates the allowed scope indicators for each directive or pragma, with `L` for loop, `R` for routine, and `G` for global. The default scope is surrounded by parentheses and N/A appears if the directive or pragma is not available in the given language.

  ### Note

  The "*" in the scope indicates this:

  For routine-scoped directive
  : The scope includes the code following the directive or pragma until the end of the routine.

  For globally-scoped directive
  : The scope includes the code following the directive or pragma until the end of the file rather than for the entire file.

  ### Note

  The name of a directive or pragma may also be prefixed with `-M`.

  For example, you can use the directive `-Mbounds`, which is equivalent to the directive `bounds` and you can use `-Mopt`, which is equivalent to `opt`. For pragmas, you can use the directive `-Mnoassoc`, which is equivalent to the pragma `noassoc`, and `-Mvintr`, which is equivalent to `vintr`.

Table 9.1. Proprietary Optimization-Related Fortran Directive and C/C++ Pragma Summary

Directive or pragma	Functionality	Default	Fortran Scope	C/C++ Scope
altcode (noaltcode)	Do/don't generate alternate code for vectorized and parallelized loops.	altcode	(L)RG	(L)RG
assoc (noassoc)	Do/don't perform associative transformations.	assoc	(L)RG	(L)RG
bounds (nobounds)	Do/don't perform array bounds checking.	nobounds	(R)G*	(R)G
cncall (nocncall)	Loops are considered for parallelization, even if they contain calls to user-defined subroutines or functions, or if their loop counts do not exceed usual thresholds.	nocncall	(L)RG	(L)RG
cncall (nocncall)	Loops are considered for parallelization, even if they contain calls to user-defined subroutines or functions, or if their loop counts do not exceed usual thresholds.	nocncall	(L)RG	(L)RG
concur (noconcur)	Do/don't enable auto-concurrentization of loops.	concur	(L)RG	(L)RG
depchk (nodepchk)	Do/don't ignore potential data dependencies.	depchk	(L)RG	(L)RG

Directive or pragma	Functionality	Default	Fortran Scope	C/C++ Scope
eqvchk (noeqvchk)	Do/don't check EQUIVALENCE for data dependencies.	eqvchk	(L)RG	N/A
fcon (nofcon)	Do/don't assume unsuffixed real constants are single precision.	nofcon	N/A	(R)G
invarif (noinvarif)	Do/don't remove invariant if constructs from loops.	invarif	(L)RG	(L)RG
ivdep	Ignore potential data dependencies.	ivdep	(L)RG	N/A
lstval (nolstval)	Do/don't compute last values.	lstval	(L)RG	(L)RG
prefetch	Control how prefetch instructions are emitted			
opt	Select optimization level.	N/A	(R)G	(R)G
safe (nosafe)	Do/don't treat pointer arguments as safe.	safe	N/A	(R)G
safe_lastval	Parallelize when loop contains a scalar used outside of loop.	not enabled	(L)	(L)
safeptr (nosafeptr)	Do/don't ignore potential data dependencies to pointers.	nosafeptr	N/A	L(R)G
single (nosingle)	Do/don't convert float parameters to double.	nosingle	N/A	(R)G*
tp	Generate PGI Unified Binary code optimized for specified targets.	N/A	(R)G	(R)G
unroll (nounroll)	Do/don't unroll loops.	nounroll	(L)RG	(L)RG
vector (novector)	Do/don't perform vectorizations.	vector	(L)RG*	(L)RG
vintr (novintr)	Do/don't recognize vector intrinsics.	vintr	(L)RG	(L)RG

Scope of Fortran Directives and Command-Line Options

During compilation the effect of a directive may be to either turn an option on, or turn an option off. Directives apply to the section of code following the directive, corresponding to the specified scope, which may include the following loop, the following routine, or the rest of the program. This section presents several examples that show the effect of directives as well as their scope. Consider the following Fortran code:

```
integer maxtime, time
parameter (n = 1000, maxtime = 10)
double precision a(n,n), b(n,n), c(n,n)
  do time = 1, maxtime
    do i = 1, n
      do j = 1, n
        c(i,j) = a(i,j) + b(i,j)
      enddo
    enddo
  enddo
end
```

When compiled with `–Mvect`, both interior loops are interchanged with the outer loop.

```
$ pgfortran -Mvect dirvect1.f
```

Directives alter this behavior either globally or on a routine or loop by loop basis. To assure that vectorization is not applied, use the `novector` directive with global scope.

```
cpgi$g novector
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

In this version, the compiler disables vectorization for the entire source file. Another use of the directive scoping mechanism turns an option on or off locally, either for a specific procedure or for a specific loop:

```
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
cpgi$l novector
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

Loop level scoping does not apply to nested loops. That is, the directive only applies to the following loop. In this example, the directive turns off vector transformations for the top-level loop. If the outer loop were a timing loop, this would be a practical use for a loop-scoped directive.

Scope of C/C++ Pragmas and Command-Line Options

During compilation a pragma either turns an option on or turns an option off. Pragmas apply to the section of code corresponding to the specified scope - either the entire file, the following loop, or the following or current routine. This section presents several examples showing the effect of pragmas and the use of the pragma scope indicators.

Note

In all cases, pragmas override a corresponding command-line option.

For pragmas that have only routine and global scope, there are two rules for determining the scope of the pragma. We cover these special scope rules at the end of this section.

Consider the following program:

```
main() {
   float a[100][100], b[100][100], c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
   n=100;
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

When this is compiled using the `-Mvect` command-line option, both interior loops are interchanged with the outer loop. Pragmas alter this behavior either globally or on a routine or loop by loop basis. To ensure that vectorization is not applied, use the `novector` pragma with global scope.

```
main() {
#pragma global novector
   float a[100][100], b[100][100],c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
   n=100;
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

In this version, the compiler does not perform vectorization for the entire source file. Another use of the pragma scoping mechanism turns an option on or off locally either for a specific procedure or for a specific loop. The following example shows the use of a loop-scoped pragma.

```
main() {
   float a[100][100], b[100][100],c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
   n=100;
#pragma loop novector
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

Loop level scoping does not apply to nested loops. That is, the pragma only applies to the following loop. In this example, the pragma turns off vector transformations for the top-level loop. If the outer loop were a timing loop, this would be a practical use for a loop-scoped pragma. The following example shows routine pragma scope:

```
#include "math.h"
func1() {
#pragma routine novector
 float a[100][100], b[100][100];
 float c[100][100], d[100][100];
 int i,j;
 for (i=0;i<100;i++)
   for (j=0;j<100;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
```

```
func2() {
 float a[200][200], b[200][200];
 float c[200][200], d[200][200];
 int i,j;
 for (i=0;i<200;i++)
   for (j=0;j<200;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
```

When this source is compiled using the `-Mvect` command-line option, func2 is vectorized but func1 is not vectorized. In the following example, the global novector pragma turns off vectorization for the entire file.

```
#include "math.h"
func1() {
#pragma global novector
 float a[100][100], b[100][100];
 float c[100][100], d[100][100];
 int i,j;
 for (i=0;i<100;i++)
   for (j=0;j<100;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
func2() {
 float a[200][200], b[200][200];
 float c[200][200], d[200][200];
 int i,j;
 for (i=0;i<200;i++)
   for (j=0;j<200;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
```

Special Scope Rules

Special rules apply for a pragma with loop, routine, and global scope. When the pragma is placed within a routine, it applies to the routine from its point in the routine to the end of the routine. The same rule applies for one of these pragmas with global scope.

However, there are several pragmas for which only routine and global scope applies and which affect code immediately following the pragma:

- bounds and fcon – The bounds and fcon pragmas behave in a similar manner to pragmas with loop scope. That is, they apply to the code following the pragma.
- opt and safe – When the opt or safe pragmas are placed within a routine, they apply to the entire routine as if they had been placed at the beginning of the routine.

Prefetch Directives and Pragmas

Today's processors are so fast that it is difficult to bring data into them quickly enough to keep them busy. Prefetch instructions can increase the speed of an application substantially by bringing data into cache so that it is available when the processor needs it.

When vectorization is enabled using the `-Mvect` or `-Mprefetch` compiler options, or an aggregate option such as `-fast` that incorporates `-Mvect`, the PGI compilers selectively emit instructions to explicitly prefetch

data into the data cache prior to first use. You can control how these prefetch instructions are emitted by using prefetch directives and pragmas.

For a list of processors that support prefetch instructions refer to the PGI Release Notes.

Prefetch Directive Syntax

The syntax of a prefetch directive is as follows:

```
  c$mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

Prefetch Directive Format Requirements

Note

The sentinel for prefetch directives is `c$mem`, which is distinct from the `cpgi$` sentinel used for optimization directives. Any prefetch directives that use the `cpgi$` sentinel are ignored by the PGI compilers.

- The "c" must be in column 1.
- Either * or ! is allowed in place of c.
- The scope indicators g, r and l used with the `cpgi$` sentinel are not supported.
- The directive name, including the directive prefix, may contain upper or lower case letters and is case insensitive (case is not significant).
- If the command line option `-Mupcase` is used, any variable names that appear in the body of the directive are case sensitive.

Sample Usage of Prefetch Directive

Example 9.1. Prefetch Directive Use

This example uses prefetch directives to prefetch data in a matrix multiplication inner loop where a row of one source matrix has been gathered into a contiguous vector.

```
 real*8 a(m,n), b(n,p), c(m,p), arow(n)
...
 do j = 1, p
c$mem prefetch arow(1),b(1,j)
c$mem prefetch arow(5),b(5,j)
c$mem prefetch arow(9),b(9,j)
 do k = 1, n, 4
c$mem prefetch arow(k+12),b(k+12,j)
 c(i,j) = c(i,j) + arow(k) * b(k,j)
 c(i,j) = c(i,j) + arow(k+1) * b(k+1,j)
 c(i,j) = c(i,j) + arow(k+2) * b(k+2,j)
 c(i,j) = c(i,j) + arow(k+3) * b(k+3,j)
 enddo
 enddo
```

This pattern of prefetch directives the compiler emits prefetch instructions whereby elements of `arow` and `b` are fetched into the data cache starting four iterations prior to first use. By varying the prefetch distance in this way, it is sometimes possible to reduce the effects of main memory latency and improve performance.

Prefetch Pragma Syntax

The syntax of a prefetch pragma is as follows:

```
#pragma mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

Sample Usage of Prefetch Pragma

Example 9.2. Prefetch Pragma in C

This example uses the prefetch pragma to prefetch data from the source vector x for eight iterations beyond the current iteration.

```
 for (i=0; i<n; i++)  {
   #pragma mem prefetch x[i+8]
   y[i] = y[i] + a*x[i];
}
```

C$PRAGMA C

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, and OSX systems, an underscore is appended to Fortran global names, including names of functions, subroutines, and common blocks. This mechanism distinguishes Fortran name space from C/C++ name space.

You can use C$PRAGMA C in the Fortran program to call a C/C++ function from Fortran. The statement would look similar to this:

```
C$PRAGMA C(name[,name]...)
```

NOTE

This statement directs the compiler to recognize the routine 'name' as a C function, thus preventing the Fortran compiler from appending an underscore to the routine name.

On Win32 systems the C$PRAGMA C as well as the attributes C and STDCALL may effect other changes on argument passing as well as on the names of the routine. For more information on this topic, refer to "Win32 Calling Conventions," on page 167.

IGNORE_TKR Directive

This directive indicates to the compiler to ignore the type, kind, and/or rank (/TKR/) of the specified dummy arguments in an interface of a procedure. The compiler also ignores the type, kind, and/or rank of the actual arguments when checking all the specifics in a generic call for ambiguities.

IGNORE_TKR Directive Syntax

The syntax for the IGNORE_TKR directive is this:

```
!DIR$ IGNORE_TKR [ [(<letter>) <dummy_arg>] ... ]
```

<letter>
is one or any combination of the following:

T - type **K** - kind **R** - rank

For example, KR indicates to ignore both kind and rank rules and TKR indicates to ignore the type, kind, and rank arguments.

<dummy_arg>
if specified, indicates the dummy argument for which TKR rules should be ignored. If not specified, TKR rules are ignored for all dummy arguments in the procedure that contains the directive.

IGNORE_TKR Directive Format Requirements

The following rules apply to this directive:

- IGNORE_TKR must not specify dummy arguments that are allocatable, Fortran 90 pointers, or assumed-shape arrays.
- IGNORE_TKR may only appear in the body of an interface block and may specify dummy argument names only.
- IGNORE_TKR may appear before or after the declarations of the dummy arguments it specifies.
- If dummy argument names are specified, IGNORE_TKR applies only to those particular dummy arguments.
- If no dummy argument names are specified, IGNORE_TKR applies to all dummy arguments except those that are allocatable objects, Fortran 90 pointers, or assumed-shape arrays.

Sample Usage of IGNORE_TKR Directive

Consider this subroutine fragment:

```
subroutine example(A,B,C,D)
!DIR$ IGNORE_TKR A, (R) B, (TK) C, (K) D
```

Table 9.2 indicates which rules are ignored for which dummy arguments in the preceding sample subroutine fragment:

Table 9.2. IGNORE_TKR Example

Dummy Argument	Ignored Rules		Dummy Argument	Ignored Rules
A	Type, Kind and Rank		C	Type and Kind
B	Only rank		D	Only Kind

Notice that no letters were specified for A, so all type, kind, and rank rules are ignored.

!DEC$ Directives

PGI Fortran compilers for Microsoft Windows support several de-facto standard Fortran directives that help with inter-language calling and importing and exporting routines to and from DLLs. These directives all take the form:

```
!DEC$ directive
```

Format Requirements

You must follow the following format requirements for the directive to be recognized in your program:

- The directive must begin in line 1 when the file is fixed format or compiled with `-Mfixed`.
- The directive prefix `!DEC$` requires a space between the prefix and the directive keyword, such as `ATTRIBUTES`.
- The ! must begin the prefix when compiling Fortran 90/95 free-form format.
- The characters C or * can be used in place of ! in either form of the prefix when compiling F77-style fixed-form format.
- The directives are completely case insensitive.

Summary Table

The following table summarizes the supported !DEC$ directives. For a complete description of each directive, refer to the "!DEC$ Directives" section of the "Directives and Pragmas Reference" chapter in the PGI Compiler Reference Manual.

Table 9.3. !DEC$ Directives Summary Table

Directive	Functionality
ALIAS	Specifies an alternative name with which to resolve a routine.
ATTRIBUTES	Lets you specify properties for data objects and procedures.
DECORATE	Specifies that the name specified in the ALIAS directive should have the prefix and postfix decorations performed on it that are associated with the calling conventions that are in effect. This directive has no effect if ALIAS is not specified.
DISTRIBUTE	Tells the compiler at what point within a loop to split into two loops.

Chapter 10. Creating and Using Libraries

A library is a collection of functions or subprograms that are grouped for reference and ease of linking. This chapter discusses issues related to PGI-supplied compiler libraries. Specifically, it addresses the use of C/C++ builtin functions in place of the corresponding libc routines, creation of dynamically linked libraries, known as shared objects or shared libraries, and math libraries.

> **Note**
>
> This chapter does not duplicate material related to using libraries for inlining, described in "Creating an Inline Library," on page 49 or information related to runtime library routines available to OpenMP programmers, described in "Runtime Library Routines," on page 63.

PGI provides libraries that export C interfaces by using Fortran modules. On Windows, PGI also provides additions to the supported library functionality for runtime functions included in DFLIB.

This chapter has examples that include the following options related to creating and using libraries.

`-Bdynamic`	`-def<file>`	`-implib <file>`	`-Mmakeimplib`
`-Bstatic`	`-dynamiclib`	`-l`	`-o`
`-c`	`-fpic`	`-Mmakedll`	`-shared`

Using builtin Math Functions in C/C++

The name of the math header file is `math.h`. Include the math header file in all of your source files that use a math library routine as in the following example, which calculates the inverse cosine of pi/3.

```
#include <math.h>
#define PI 3.1415926535
void main()
{
  double x, y;
  x = PI/3.0;
  y = acos(x);
}
```

Including `math.h` causes PGCC C and C++ to use builtin functions, which are much more efficient than library calls. In particular, if you include `math.h`, the following intrinsics calls are processed using builtins:

abs	acosf	asinf	atan	atan2	atan2f
atanf	cos	cosf	exp	expf	fabs
fabsf	fmax	fmaxf	fmin	fminf	log
log10	log10f	logf	pow	powf	sin
sinf	sqrt	sqrtf	tan	tanf	

Using System Library Routines

Release 13.10 of the PGI runtime libraries makes use of Linux system libraries to implement, for example, OpenMP and Fortran I/O. The PGI runtime libraries make use of several additional system library routines.

On 64-bit Linux systems, the system library routines that PGI supports include these:

aio_error	aio_write	pthread_mutex_init	sleep
aio_read	calloc	pthread_mutex_lock	
aio_return	getrlimit	pthread_mutex_unlock	
aio_suspend	pthread_attr_init	setrlimit	

On 32-bit Linux systems, the system library routines that PGI supports include these:

aio_error	aio_suspend	getrlimit	sleep
aio_read	aio_write	pthread_attr_init	
aio_return	calloc	setrlimit	

Creating and Using Shared Object Files on Linux

All of the PGI Fortran, C, and C++ compilers support creation of shared object files. Unlike statically-linked object and library files, shared object files link and resolve references with an executable at runtime via a dynamic linker supplied with your operating system. The PGI compilers must generate position independent code to support creation of shared objects by the linker. However, this is not the default. You must create object files with position independent code and shared object files that will include them.

Procedure to create a use a shared object file

The following steps describe how to create and use a shared object file.

1. Create an object file with position independent code.

   To do this, compile your code with the appropriate PGI compiler using the `–fpic` option, or one of the equivalent options, such as `–fPIC`, `–Kpic,` and `–KPIC`, which are supported for compatibility with other systems. For example, use the following command to create an object file with position independent code using pgfortran:

   ```
   % pgfortran -c -fpic tobeshared.f
   ```

2. Produce a shared object file.

To do this, use the appropriate PGI compiler to invoke the linker supplied with your system. It is customary to name such files using a `.so` filename extension. On Linux, you do this by passing the `–shared` option to the linker:

```
% pgfortran -shared -o tobeshared.so tobeshared.o
```

Note

Compilation and generation of the shared object can be performed in one step using both the `–fpic` option and the appropriate option for generation of a shared object file.

3. Use a shared object file.

   To do this, use the appropriate PGI compiler to compile and link the program which will reference functions or subroutines in the shared object file, and list the shared object on the link line, as shown here:

   ```
   % pgfortran -o myprog myprog.f tobeshared.so
   ```

4. Make the executable available.

   You now have an executable `myprog` which does not include any code from functions or subroutines in `tobeshared.so`, but which can be executed and dynamically linked to that code. By default, when the program is linked to produce `myprog`, no assumptions are made on the location of `tobeshared.so`. Therefore, for `myprog` to execute correctly, you must initialize the environment variable `LD_LIBRARY_PATH` to include the directory containing `tobeshared.so`. If `LD_LIBRARY_PATH` is already initialized, it is important not to overwrite its contents. Assuming you have placed `tobeshared.so` in a directory `/home/myusername/bin`, you can initialize `LD_LIBRARY_PATH` to include that directory and preserve its existing contents, as shown in the following:

   ```
   % setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":/home/myusername/bin
   ```

   If you know that `tobeshared.so` always resides in a specific directory, you can create the executable `myprog` in a form that assumes this directory by using the `–R` link-time option. For example, you can link as follows:

   ```
   % pgfortran -o myprog myprof.f tobeshared.so -R/home/myusername/bin
   ```

   #### Note

   As with the `–L` option, there is no space between `–R` and the directory name. If the –R option is used, it is not necessary to initialize `LD_LIBRARY_PATH`.

   In the previous example, the dynamic linker always looks in `/home/myusername/bin` to resolve references to `tobeshared.so`. By default, if the `LD_LIBRARY_PATH` environment variable is not set, the linker only searches `/usr/lib` and `/lib` for shared objects.

ldd Command

The `ldd` command is a useful tool when working with shared object files and executables that reference them. When applied to an executable, as shown in the following example, `ldd` lists all shared object files referenced in the executable along with the pathname of the directory from which they will be extracted.

```
% ldd myprog
```

If the pathname is not hard-coded using the `-R` option, and if `LD_LIBRARY_PATH` is not initialized, the pathname is listed as "not found". For more information on `ldd`, its options and usage, see the online man page for `ldd`.

Creating and Using Dynamic Libraries on Mac OS X

Note

PGI compilers for Mac OS X do not support static linking on user executables. Apple only ships dynamic versions of its system libraries - not static versions. You can create static libraries; however, you cannot create 100% static executables.

The 32-bit version of PGI Workstation for Mac OS X supports generation of dynamic libraries. To create the dynamic library, you use the `-dynamiclib` switch to invoke the libtool utility program provided by Mac OS X. For more information, refer to the `libtool` man page.

The following example creates and uses a dynamic library:

1. Create the object files.

   `world.f90:`

   ```
   subroutine world
   print *, 'Hello World!'
   end
   ```

   `hello.f90:`

   ```
   program hello
   call world
   end
   ```

2. Build the dynamic library:

   ```
   % pgfortran -dynamiclib world.f90 -o world.dylib
   ```

3. Build the program that uses the dynamic library:

   ```
   % pgfortran hello.f90 world.dylib -o hello
   ```

4. Run the program:

   ```
   % ./hello|
    Hello World!
   ```

PGI Runtime Libraries on Windows

The PGI runtime libraries on Windows are available in both static and dynamically-linked (DLL) versions. The static libraries are used by default.

- You can use the dynamically-linked version of the runtime by specifying `-Bdynamic` at both compile and link time.

  ### Note

  C++ on Windows does not support `-Bdynamic`.

- You can explicitly specify static linking, the default, by using `-Bstatic` at compile and link time.

For details on why you might choose one type of linking over another type, refer to "Creating and Using Dynamic-Link Libraries on Windows," on page 124.

Creating and Using Static Libraries on Windows

The Microsoft Library Manager (`LIB.EXE`) is the tool that is typically used to create and manage a static library of object files on Windows. `LIB` is provided with the PGI compilers as part of the Microsoft Open Tools. Refer to *www.msdn2.com* for a complete `LIB` reference - search for `LIB.EXE`. For a list of available options, invoke `LIB` with the `/?` switch.

For compatibility with legacy makefiles, PGI provides a wrapper for `LIB` and `LINK` called `ar`. This version of `ar` is compatible with Windows and object-file formats.

PGI also provides `ranlib` as a placeholder for legacy makefile support.

ar command

The `ar` command is a legacy archive wrapper that interprets legacy `ar` command line options and translates these to `LINK/LIB` options. You can use it to create libraries of object files.

Syntax:

The syntax for the `ar` command is this:

```
ar [options] [archive] [object file].
```

Where:

- The first argument must be a command line switch, and the leading dash on the first option is optional.
- The single character options, such as `-d` and `-v`, may be combined into a single option, as `-dv`.

  Thus, `ar dv`, `ar -dv,` and `ar -d -v` all mean the same thing.
- The first non-switch argument must be the library name.
- One (and only one) of `-d`, `-r`, `-t`, or `-x` must appear on the command line.

Options

The options available for the `ar` command are these:

–c
: This switch is for compatibility; it is ignored.

–d
: Deletes the named object files from the library.

–r
: Replaces in or adds the named object files to the library.

–t
Writes a table of contents of the library to standard out.

–v
Writes a verbose file-by-file description of the making of the new library to standard out.

–x
Extracts the named files by copying them into the current directory.

ranlib command

The `ranlib` command is a wrapper that allows use of legacy scripts and makefiles that use the `ranlib` command. The command actually does nothing; it merely exists for compatibility.

Syntax:

The syntax for the `ranlib` command is this:

```
ranlib [options] [archive]
```

Options

The options available for the `ranlib` command are these:

–help
Short help information is printed out.

–V
Version information is printed out.

Creating and Using Dynamic-Link Libraries on Windows

There are several differences between static and dynamic-link libraries on Windows. Libraries of either type are used when resolving external references for linking an executable, but the process differs for each type of library. When linking with a static library, the code needed from the library is incorporated into the executable. When linking with a DLL, external references are resolved using the DLL's import library, not the DLL itself. The code in the DLL associated with the external references does not become a part of the executable. The DLL is loaded when the executable that needs it is run. For the DLL to be loaded in this manner, the DLL must be in your path.

Static libraries and DLLs also handle global data differently. Global data in static libraries is automatically accessible to other objects linked into an executable. Global data in a DLL can only be accessed from outside the DLL if the DLL exports the data and the image that uses the data imports it.

To access global data, the C compilers support the Microsoft storage class extensions: `__declspec(dllimport)` and `__declspec(dllexport)`. These extensions may appear as storage class modifiers and enable functions and data to be imported and exported:

```
extern int __declspec(dllimport) intfunc();
float __declspec(dllexport) fdata;
```

The PGI Fortran compilers support the DEC$ ATTRIBUTES extensions `DLLIMPORT` and `DLLEXPORT`:

```
cDEC$ ATTRIBUTES DLLEXPORT :: object [,object] ...
```

```
cDEC$ ATTRIBUTES DLLIMPORT :: object [,object] ...
```

Here *c* is one of C, c, !, or *. `object` is the name of the subprogram or common block that is exported or imported. Further, common block names are enclosed within slashes (/), as shown here:

```
cDEC$ ATTRIBUTES DLLIMPORT :: intfunc
!DEC$ ATTRIBUTES DLLEXPORT :: /fdata/
```

For more information on these extensions, refer to "!DEC$ Directives," on page 117.

The examples in this section further illustrate the use of these extensions.

To create a DLL from the command line, use the `-Mmakedll` option.

The following switches apply to making and using DLLs with the PGI compilers:

–Bdynamic
: Compile for and link to the DLL version of the PGI runtime libraries. This flag is required when linking with any DLL built by the PGI compilers. This flag corresponds to the `/MD` flag used by Microsoft's `cl` compilers.

  When you use the PGI compiler flag `-Bdynamic` to create an executable that links to the DLL form of the runtime, the executable built is smaller than one built without `-Bdynamic`. The PGI runtime DLLs, however, must be available on the system where the executable is run. You must use the `-Bdynamic` flag when linking an executable against a DLL built by the PGI compilers.

  Note

  C++ on Windows does not support `-Bdynamic`.

–Bstatic
: Compile for and link to the static version of the PGI runtime libraries. This flag corresponds to the `/MT` flag used by Microsoft's `cl` compilers.

  On Windows, you must use`-Bstatic` for both compiling and linking.

–Mmakedll
: Generate a dynamic-link library or DLL. Implies `-Bdynamic`.

–Mmakeimplib
: Generate an import library without generating a DLL. Use this flag when you want to generate an import library for a DLL but are not yet ready to build the DLL itself. This situation might arise, for example, when building DLLs with mutual imports, as shown in Example 10.4, "Build DLLs Containing Mutual Imports: Fortran," on page 129.

–o <file>
: Passed to the linker. Name the DLL or import library <file>.

–def <file>
: When used with `-Mmakedll`, this flag is passed to the linker and a `.def` file named <file> is generated for the DLL. The `.def` file contains the symbols exported by the DLL. Generating a `.def` file is not required when building a DLL but can be a useful debugging tool if the DLL does not contain the symbols that you expect it to contain.

When used with –Mmakeimplib, this flag is passed to lib which requires a .def file to create an import library. The .def file can be empty if the list of symbols to export are passed to lib on the command line or explicitly marked as DLLEXPORT in the source code.

–implib <file>
Passed to the colinker. Generate an import library named <file> for the DLL. A DLL's import library is the interface used when linking an executable that depends on routines in a DLL.

To use the PGI compilers to create an executable that links to the DLL form of the runtime, use the compiler flag –Bdynamic. The executable built will be smaller than one built without –Bdynamic; the PGI runtime DLLs, however, must be available on the system where the executable is run. The –Bdynamic flag must be used when an executable is linked against a DLL built by the PGI compilers.

The following examples outline how to use –Bdynamic, –Mmakedll and –Mmakeimplib to build and use DLLs with the PGI compilers.

Note

C++ on Windows does not support –Bdynamic.

Example 10.1. Build a DLL: Fortran

This example builds a DLL from a single source file, object1.f, which exports data and a subroutine using DLLEXPORT. The source file, prog1.f, uses DLLIMPORT to import the data and subroutine from the DLL.

object1.f

```
 subroutine sub1(i)
!DEC$ ATTRIBUTES DLLEXPORT :: sub1
 integer i
 common /acommon/ adata
 integer adata
!DEC$ ATTRIBUTES DLLEXPORT :: /acommon/
 print *, "sub1 adata", adata
 print *, "sub1 i ", i
 adata = i
 end
```

prog1.f

```
 program prog1
 common /acommon/ adata
 integer adata
 external sub1
!DEC$ ATTRIBUTES DLLIMPORT:: sub1, /acommon/
 adata = 11
 call sub1(12)
 print *, "main adata", adata
 end
```

Step 1: Create the DLL obj1.dll and its import library obj1.lib using the following series of commands:

```
% pgfortran -Bdynamic -c object1.f
% pgfortran -Mmakedll object1.obj -o obj1.dll
```

Step 2: Compile the main program:

```
% pgfortran -Bdynamic -o prog1 prog1.f -defaultlib:obj1
```

The `-Bdynamic` and `-Mmakedll` switches cause the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `-Bdynamic` switch is required when linking against any PGI-compiled DLL, such as `obj1.dll`. The -defaultlib: switch specifies that `obj1.lib`, the DLL's import library, should be used to resolve imports.

Step 3: Ensure that `obj1.dll` is in your path, then run the executable `prog1` to determine if the DLL was successfully created and linked:

```
% prog1
sub1 adata 11
sub1 i 12
main adata 12
```

Should you wish to change `obj1.dll` without changing the subroutine or function interfaces, no rebuilding of `prog1` is necessary. Just recreate `obj1.dll` and the new `obj1.dll` is loaded at runtime.

Example 10.2. Build a DLL: C

In this example, we build a DLL out of a single source file, `object2.c`, which exports data and a subroutine using `__declspec(dllexport)`. The main source file, `prog2.c`, uses `__declspec(dllimport)` to import the data and subroutine from the DLL.

`object2.c`

```
int __declspec(dllexport) data;
void __declspec(dllexport)
func2(int i)
{
  printf("func2: data == %d\n", data);
  printf("func2: i == %d\n", i);
  data = i;
}
```

`prog2.c`

```
int __declspec(dllimport) data;
void __declspec(dllimport) func2(int);
int
main()
{
  data = 11;
  func2(12);
  printf("main: data == %d\n",data);
  return 0;
}
```

Step 1: Create the DLL `obj2.dll` and its import library `obj2.lib` using the following series of commands:

```
% pgcc -Bdynamic -c object2.c
% pgcc -Mmakedll object2.obj -o obj2.dll
```

Step 2: Compile the main program:

```
% pgcc -Bdynamic -o prog2 prog2.c -defaultlib:obj2
```

The `-Bdynamic` switch causes the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `-Bdynamic` switch is required when linking against any PGI-compiled DLL such as `obj2.dll`. The `#defaultlib:` switch specifies that `obj2.lib`, the DLL's import library, should be used to resolve the imported data and subroutine in `prog2.c`.

Step 3: Ensure that `obj2.dll` is in your path, then run the executable `prog2` to determine if the DLL was successfully created and linked:

```
% prog2
func2: data == 11
func2: i == 12
main: data == 12
```

Should you wish to change `obj2.dll` without changing the subroutine or function interfaces, no rebuilding of `prog2` is necessary. Just recreate `obj2.dll` and the new `obj2.dll` is loaded at runtime.

Example 10.3. Build DLLs Containing Circular Mutual Imports: C

In this example we build two DLLs, `obj3.dll` and `obj4.dll`, each of which imports a routine that is exported by the other. To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation. Once the DLLs are built, we can use them to build the main program.

```
/* object3.c */

 void __declspec(dllimport) func_4b(void);
 void __declspec(dllexport)
 func_3a(void)
 {
   printf("func_3a, calling a routine in obj4.dll\n");
   func_4b();
 }
 void __declspec(dllexport)
 func_3b(void)
 {
   printf("func_3b\n");
 }
```

```
/* object4.c */
 void __declspec(dllimport) func_3b(void);
 void __declspec(dllexport)
 func_4a(void)
 {
   printf("func_4a, calling a routine in obj3.dll\n");
   func_3b();
 }
 void __declspec(dllexport)
 func_4b(void)
 {
   printf("func_4b\n");
 }
```

```
/* prog3.c */
 void __declspec(dllimport) func_3a(void);
 void __declspec(dllimport) func_4a(void);
 int
```

```
main()
{
  func_3a();
  func_4a();
  return 0;
}
```

Step 1: Use `-Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

```
% pgcc -Bdynamic -c object3.c
% pgcc -Mmakeimplib -o obj3.lib object3.obj
```

Tip

The `-def=<deffile>` option can also be used with `-Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `__declspec(dllexport)`.

Step 2: Use the import library, `obj3.lib`, created in Step 1, to link the second DLL.

```
% pgcc -Bdynamic -c object4.c
% pgcc -Mmakedll -o obj4.dll object4.obj -defaultlib:obj3
```

Step 3: Use the import library, `obj4.lib`, created in Step 2, to link the first DLL.

```
% pgcc -Mmakedll -o obj3.dll object3.obj -defaultlib:obj4
```

Step 4: Compile the main program and link against the import libraries for the two DLLs.

```
% pgcc -Bdynamic prog3.c -o prog3 -defaultlib:obj3 -defaultlib:obj4
```

Step 5: Execute `prog3.exe` to ensure that the DLLs were create properly.

```
% prog3
func_3a, calling a routine in obj4.dll
func_4b
func_4a, calling a routine in obj3.dll
func_3b
```

Example 10.4. Build DLLs Containing Mutual Imports: Fortran

In this example we build two DLLs when each DLL is dependent on the other, and use them to build the main program.

In the following source files, `object2.f95` makes calls to routines defined in `object3.f95`, and vice versa. This situation of mutual imports requires two steps to build each DLL.

To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation.

Once the DLLs are built, we can use them to build the main program.

object2.f95

```
 subroutine func_2a
 external func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3b
 print*,"func_2a, calling a routine in obj3.dll"
 call func_3b()
 end subroutine
```

```
 subroutine func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2b
 print*,"func_2b"
 end subroutine
```

object3.f95

```
 subroutine func_3a
 external func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2b
 print*,"func_3a, calling a routine in obj2.dll"
 call func_2b()
 end subroutine
```

```
 subroutine func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3b
 print*,"func_3b"
 end subroutine
```

prog2.f95

```
 program prog2
 external func_2a
 external func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3a
 call func_2a()
 call func_3a()
 end program
```

Step 1: Use `-Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

```
% pgfortran -Bdynamic -c object2.f95
% pgfortran -Mmakeimplib -o obj2.lib object2.obj
```

Tip

The `-def=<deffile>` option can also be used with `-Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `DLLEXPORT`.

Step 2: Use the import library, `obj2.lib`, created in Step 1, to link the second DLL.

```
% pgfortran -Bdynamic -c object3.f95
% pgfortran -Mmakedll -o obj3.dll object3.obj -defaultlib:obj2
```

Step 3: Use the import library, `obj3.lib`, created in Step 2, to link the first DLL.

```
% pgfortran -Mmakedll -o obj2.dll object2.obj -defaultlib:obj3
```

Step 4: Compile the main program and link against the import libraries for the two DLLs.

```
% pgfortran -Bdynamic prog2.f95 -o prog2 -defaultlib:obj2 -defaultlib:obj3
```

Step 5: Execute `prog2` to ensure that the DLLs were created properly:

```
% prog2
func_2a, calling a routine in obj3.dll
func_3b
func_3a, calling a routine in obj2.dll
func_2b
```

Example 10.5. Import a Fortran module from a DLL

In this example we import a Fortran module from a DLL. We use the source file `defmod.f90` to create a DLL containing a Fortran module. We then use the source file `use_mod.f90` to build a program that imports and uses the Fortran module from `defmod.f90`.

defmod.f90

```
module testm
 type a_type
 integer :: an_int
 end type a_type
 type(a_type) :: a, b
!DEC$ ATTRIBUTES DLLEXPORT :: a,b
 contains
 subroutine print_a
!DEC$ ATTRIBUTES DLLEXPORT :: print_a
 write(*,*) a%an_int
 end subroutine
 subroutine print_b
!DEC$ ATTRIBUTES DLLEXPORT :: print_b
 write(*,*) b%an_int
 end subroutine
end module
```

usemod.f90

```
use testm
 a%an_int = 1
 b%an_int = 2
 call print_a
 call print_b
end
```

Step 1: Create the DLL.

```
% pgf90 -Mmakedll -o defmod.dll defmod.f90
Creating library defmod.lib and object defmod.exp
```

Step 2: Create the `exe` and link against the import library for the imported DLL.

```
% pgf90 -Bdynamic -o usemod usemod.f90 -defaultlib:defmod.lib
```

Step 3: Run the `exe` to ensure that the module was imported from the DLL properly.

```
% usemod
1
2
```

Using LIB3F

The PGI Fortran compilers include complete support for the de facto standard LIB3F library routines on both Linux and Windows operating systems. See the PGI Fortran Language Reference manual for a complete list of available routines in the PGI implementation of LIB3F.

LAPACK, BLAS and FFTs

Pre-compiled versions of the public domain LAPACK and BLAS libraries are included with the PGI compilers. The LAPACK library is called `liblapack.a` or on Windows, `liblapack.lib`. The BLAS library is called `libblas.a` or on Windows, `libblas.lib`. These libraries are installed to `$PGI/<target>/lib`, where <target> is replaced with the appropriate target name (linux86, linux86-64, osx86, osx86-64, win32, or win64.

To use these libraries, simply link them in using the `-l` option when linking your main program:

```
% pgfortran myprog.f -llapack -lblas
```

Highly optimized assembly-coded versions of BLAS and certain FFT routines may be available for your platform. In some cases, these are shipped with the PGI compilers. See the current release notes for the PGI compilers you are using to determine if these optimized libraries exist, where they can be downloaded (if necessary), and how to incorporate them into your installation as the default.

Linking with ScaLAPACK

The ScaLAPACK libraries are automatically installed by the **installcdk** script described in the *PGI Installation Guide*. You can link with the ScaLAPACK libraries by specifying `-Mscalapack` on any of the *PGI CDK* compiler command lines. For example:

```
% pgf77 myprog.f -Mmpi=mpich1 -Mscalapack
```
or
```
% pgf77 myprog.f -Mmpi=mpich2 -Mscalapack
```

The `-Mscalapack` option causes the following libraries to be linked into your executable:

```
scalapack.a
blacsCinit_MPI-LINUX-0.a
blacs_MPI-LINUX-0.a
blacsF77init_MPI-LINUX-0.a
libblas.a
libmpich.a
```

These libraries are installed in

```
$PGI/linux86/13.10/mpi/mpich/lib
$PGI/linux86/13.10/mpi2/mpich/lib
$PGI/linux86/13.10/mpi/mvapich/lib.
```

You run a program that uses ScaLAPACK routines just like any other MPI program. The version of ScaLAPACK included in the *PGI CDK* is pre-configured for use with MPICH.

If you wish to use a different BLAS library, and still use the `-Mscalapack` switch, then copy your BLAS library into `$PGI/linux86/13.10/lib/libblas.a`.

Alternatively, you can just list the above set of libraries explicitly on your link line. You can test that ScaLAPACK is properly installed by running a test program as outlined in the following section.

The C++ Standard Template Library

The PGC++ compiler includes a bundled copy of the STLPort Standard C++ Library. See the online Standard C ++ Library tutorial and reference manual at *www.stlport.com* for further details and licensing.

Limitations

The Open Source Cluster utilities, in particular the MPICH and ScaLAPACK libraries, are provided with support necessary to build and define their proper use. However, use of these libraries on linux86-64 systems is subject to the following limitations:

- MPI libraries are limited to Messages of length < 2GB, and integer arguments are *INTEGER*4* in FORTRAN, and *int* in C.
- Integer arguments for ScaLAPACK libraries are *INTEGER*4* in FORTRAN, and *int* in C.
- Arrays passed must be < 2GB in size.

Chapter 11. Using Environment Variables

Environment variables allow you to set and pass information that can alter the default behavior of the PGI compilers and the executables which they generate. This chapter includes explanations of the environment variables specific to PGI compilers. Other environment variables are referenced and documented in other sections of this User's Guide, the accompanying Reference Manual, the PGDBG Debugger Guide and the PGPROF Profiler Manual.

- You use OpenMP environment variables to control the behavior of OpenMP programs. For consistency related to the OpenMP environment, the details of the OpenMP-related environment variables are included in Chapter 5, "*Using OpenMP*".
- You can use environment variables to control the behavior of the PGDBG debugger or PGPROF profiler. For a description of environment variables that affect these tools, refer to the *PGDBG Debugger Manual* and *PGPROF Profiler Manual*, respectively.

Setting Environment Variables

Before we look at the environment variables that you might use with the PGI compilers and tools, let's take a look at how to set environment variables. To illustrate how to set these variables in various environments, lets look at how a user might initialize the shell environment prior to using the PGI compilers and tools.

Setting Environment Variables on Linux

Let's assume that you want access to the PGI products when you log in. Let's further assume that you installed the PGI compilers in `/opt/pgi` and that the license file is in `/opt/pgi/license.dat`. For access at startup, you can add the following lines to your startup file.

In csh, use these commands:

```
% setenv PGI /opt/pgi
% setenv MANPATH "$MANPATH":$PGI/linux86/13.10/man
% setenv LM_LICENSE_FILE $PGI/license.dat
% set path = ($PGI/linux86/13.10/bin $path)
```

In bash, sh, zsh, or ksh, use these commands:

```
$ PGI=/opt/pgi; export PGI
$ MANPATH=$MANPATH:$PGI/linux86/13.10/man; export MANPATH
$ LM_LICENSE_FILE=$PGI/license.dat; export LM_LICENSE_FILE
$ PATH=$PGI/linux86/13.10/bin:$PATH; export PATH
```

Setting Environment Variables on Windows

In Windows, when you access PGI Workstation 13.10 (for example, using *Start* | *ALL Programs* | *PGI Workstation* | *Command Shells 13.10*), you have options that PGI provides for setting your environment variables - either the DOS command environment or the Cygwin Bash environment.

When you open either of these shells available to you, the default environment variables are already set and available to you.

You may want to use other environment variables, such as the OpenMP ones. This section explains how to do that.

Suppose that your home directory is `C:\tmp`. The following examples show how you might set the temporary directory to your home directory, and then verify that it is set.

Command prompt:

Once you have launched a command shell for the version of PGI that you are using, (32-bit or 64-bit), enter the following:

```
DOS> set TMPDIR=C:\tmp
DOS> echo %TMPDIR%
C:\tmp
DOS>
```

Cygwin Bash prompt:

From PGI Workstation 13.10, select PGI Workstation (32-bit or 64-bit) and at the Cygwin Bash prompt, enter the following

```
PGI$ export TMPDIR=C:\tmp
PGI$ echo $TMPDIR
C:\tmp
PGI$
```

Setting Environment Variables on Mac OSX

Let's assume that you want access to the PGI products when you log in. Let's further assume that you installed the PGI compilers in `/opt/pgi` and that the license file is in `/opt/pgi/license.dat`. For access at startup, you can add the following lines to your startup file.

For x64 osx86-64 in a csh:

```
% set path = (/opt/pgi/osx86-64/13.10/bin $path)
% setenv MANPATH "$MANPATH":/opt/pgi/osx86-64/13.10/man
```

For x64 osx86-64 in a bash, sh, zsh, or ksh:

```
$ PATH=/opt/pgi/osx86-64/13.10/bin:$PATH; export PATH
$ MANPATH=$MANPATH:/opt/pgi/osx86-64/13.10/man; export MANPATH
```

PGI-Related Environment Variables

For easy reference, the following table provides a quick listing of some OpenMP and all PGI compiler-related environment variables. This section provides more detailed descriptions of the environment variables specific to PGI compilers and the executables they generate. For information specific to OpenMP environment variables, refer to Table 5.4, "OpenMP-related Environment Variable Summary Table" and to the complete descriptions in "OpenMP Environment Variables" in the PGI Compiler Reference Manual.

Table 11.1. PGI-Related Environment Variable Summary

Environment Variable	Description
FLEXLM_BATCH	(Windows only) When set to 1, prevents interactive pop-ups from appearing by sending all licensing errors and warnings to standard out rather than to a pop-up window.
FORTRANOPT	Allows the user to specify that the PGI Fortran compilers user VAX I/O conventions.
GMON_OUT_PREFIX	Specifies the name of the output file for programs that are compiled and linked with the –pg option.
LD_LIBRARY_PATH	Specifies a colon-separated set of directories where libraries should first be searched, prior to searching the standard set of directories.
LM_LICENSE_FILE	Specifies the full path of the license file that is required for running the PGI software. On Windows, `LM_LICENSE_FILE` does not need to be set.
MANPATH	Sets the directories that are searched for manual pages associated with the command that the user types.
MPSTKZ	Increases the size of the stacks used by threads executing in parallel regions. The value should be an integer <n> concatenated with `M` or `m` to specify stack sizes of n megabytes.
MP_BIND	Specifies whether to bind processes or threads executing in a parallel region to a physical processor.
MP_BLIST	When `MP_BIND` is `yes`, this variable specifically defines the thread-CPU relationship, overriding the default values.
MP_SPIN	Specifies the number of times to check a semaphore before calling sched_yield() (on Linux or Mac OS X) or _sleep() (on Windows).
MP_WARN	Allows you to eliminate certain default warning messages.
NCPUS	Sets the number of processes or threads used in parallel regions.
NCPUS_MAX	Limits the maximum number of processors or threads that can be used in a parallel region.
NO_STOP_MESSAGE	If used, the execution of a plain STOP statement does not produce the message FORTRAN STOP.
OMP_DYNAMIC	Currently has no effect. Enables (`TRUE`) or disables (`FALSE`) the dynamic adjustment of the number of threads. The default is FALSE.

Environment Variable	Description
OMP_MAX_ACTIVE_LEVELS	Specifies the maximum number of nested parallel regions.
OMP_NESTED	Currently has no effect. Enables (`TRUE`) or disables (`FALSE`) nested parallelism. The default is FALSE.
OMP_NUM_THREADS	Specifies the number of threads to use during execution of parallel regions. Default is 1.
OMP_SCHEDULE	Specifies the type of iteration scheduling and, optionally, the chunk size to use for *omp for* and *omp parallel for* loops that include the runtime schedule clause. The default is STATIC with chunk size = 1.
OMP_STACKSIZE	Overrides the default stack size for a newly created thread.
OMP_WAIT_POLICY	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are `ACTIVE` and `PASSIVE`. The default is `ACTIVE`.
PATH	Determines which locations are searched for commands the user may type.
PGI	Specifies, at compile-time, the root directory where the PGI compilers and tools are installed.
PGI_CONTINUE	If set, when a program compiled with `-Mchkfpstk` is executed, the stack is automatically cleaned up and execution then continues.
PGI_OBJSUFFIX	Allows you to control the suffix on generated object files.
PGI_STACK_USAGE	(Windows only) Allows you to explicitly set stack properties for your program.
PGI_TERM	Controls the stack traceback and just-in-time debugging functionality.
PGI_TERM_DEBUG	Overrides the default behavior when `PGI_TERM` is set to `debug`.
PWD	Allows you to display the current directory.
STATIC_RANDOM_SEED	Forces the seed returned by `RANDOM_SEED` to be constant.
TMP	Sets the directory to use for temporary files created during execution of the PGI compilers and tools; interchangeable with `TMPDIR`.
TMPDIR	Sets the directory to use for temporary files created during execution of the PGI compilers and tools.

PGI Environment Variables

You use the environment variables listed in Table 11.1 to alter the default behavior of the PGI compilers and the executables which they generate. This section provides more detailed descriptions about the variables in this table that are not OpenMP environment variables.

FLEXLM_BATCH

By default, on Windows the license server creates interactive pop-up messages to issue warning and errors. You can use the environment variable FLEXLM_BATCH to prevent interactive pop-up windows. To do this, set the environment variable FLEXLM_BATCH to 1.

The following csh example prevents interactive pop-up messages for licensing warnings and errors:

```
% set FLEXLM_BATCH = 1;
```

FORTRANOPT

FORTRANOPT allows the user to adjust the behavior of the PGI Fortran compilers.

- If FORTRANOPT exists and contains the value vaxio, the record length in the open statement is in units of 4-byte words, and the $ edit descriptor only has an effect for lines beginning with a space or a plus sign (+).
- If FORTRANOPT exists and contains the value format_relaxed, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) is not required to be a type implied by the descriptor.
- In a non-Windows environment, if FORTRANOPT exists and contains the value crif, a sequential formatted or list-directed record is allowed to be terminated with the character sequence \r\n (carriage return, newline). This approach is useful when reading records from a file produced on a Window's system.

The following example causes the PGI Fortran compilers to use VAX I/O conventions:

```
% setenv FORTRANOPT vaxio
```

GMON_OUT_PREFIX

GMON_OUT_PREFIX specifies the name of the output file for programs that are compiled and linked with the -pg option. The default name is gmon.out.a.

If GMON_OUT_PREFIX is set, the name of the output file has GMON_OUT_PREFIX as a prefix. Further, the suffix is the pid of the running process. The prefix and suffix are separated by a dot. For example, if the output file is mygmon, then the full filename may look something similar to this: GMON_OUT_PREFIX.mygmon.0012348567.

The following example causes the PGI Fortran compilers to use pgout as the output file for programs compiled and linked with the -pg option.

```
% setenv GMON_OUT_PREFIX pgout
```

LD_LIBRARY_PATH

The LD_LIBRARY_PATH variable is a colon-separated set of directories specifying where libraries should first be searched, prior to searching the standard set of directories. This variable is useful when debugging a new library or using a nonstandard library for special purposes.

The following csh example adds the current directory to your LD_LIBRARY_PATH variable.

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":"./"
```

LM_LICENSE_FILE

The LM_LICENSE_FILE variable specifies the full path of the license file that is required for running the PGI software.

For example, once the license file is in place, you can execute the following csh commands to make the products you have purchased accessible and to initialize your environment for use of FLEXlm. These commands assume that you use the default installation directory: /opt/pgi

```
% setenv PGI /opt/pgi
% setenv LM_LICENSE_FILE "$LM_LICENSE_FILE":/opt/pgi/license.dat
```

To set the environment variable LM_LICENSE_FILE to the full path of the license key file, do this:

1. Open the System Properties dialog: *Start* | *Control Panel* | *System*.
2. Select the *Advanced* tab.
3. Click the *Environment Variables* button.
   - If LM_LICENSE_FILE is not already an environment variable, create a new system variable for it. Set its value to the full path, including the name of the license key file, license.dat.
   - If LM_LICENSE_FILE already exists as an environment variable, append the path to the license file to the variable's current value using a semi-colon to separate entries.

MANPATH

The MANPATH variable sets the directories that are searched for manual pages associated with the commands that the user types. When using PGI products, it is important that you set your PATH to include the location of the PGI products and then set the MANPATH variable to include the man pages associated with the products.

The following csh example targets x64 linux86-64 version of the compilers and tool s and allows the user access to the manual pages associated with them.

```
% set path = (/opt/pgi/linux86-64/13.10/bin $path
% setenv MANPATH "$MANPATH":/opt/pgi/linux86-64/13.10/man
```

MPSTKZ

MPSTKZ increases the size of the stacks used by threads executing in parallel regions. You typically use this variable with programs that utilize large amounts of thread-local storage in the form of private variables or local variables in functions or subroutines called within parallel regions. The value should be an integer <n> concatenated with M or m to specify stack sizes of n megabytes.

For example, the following setting specifies a stack size of 8 megabytes.

```
% setenv MPSTKZ 8M
```

MP_BIND

You can set MP_BIND to yes or y to bind processes or threads executing in a parallel region to physical processor. Set it to no or n to disable such binding. The default is to not bind processes to processors. This

variable is an execution-time environment variable interpreted by the PGI runtime support libraries. It does not affect the behavior of the PGI compilers in any way.

Note

The MP_BIND environment variable is not supported on all platforms.

```
% setenv MP_BIND y
```

MP_BLIST

MP_BLIST allows you to specifically define the thread-CPU relationship.

Note

This variable is only in effect when MP_BIND is yes.

While the MP_BIND variable binds processors or threads to a physical processor, MP_BLIST allows you to specifically define which thread is associated with which processor. The list defines the processor-thread relationship order, beginning with thread 0. This list overrides the default binding.

For example, the following setting for MP_BLIST maps CPUs 3, 2, 1 and 0 to threads 0, 1, 2 and 3 respectively.

```
% setenv MP_BLIST=3,2,1,0
```

MP_SPIN

When a thread executing in a parallel region enters a barrier, it spins on a semaphore. You can use MP_SPIN to specify the number of times it checks the semaphore before calling sched_yield() (on Linux or MAC OS X) or _sleep() (on Windows). These calls cause the thread to be re-scheduled, allowing other processes to run. The default value is 1000000..

```
% setenv MP_SPIN 200
```

MP_WARN

MP_WARN allows you to eliminate certain default warning messages.

By default, a warning is printed to stderr if you execute an OpenMP or auto-parallelized program with NCPUS or OMP_NUM_THREADS set to a value larger than the number of physical processors in the system.

For example, if you produce a parallelized executable a.out and execute as follows on a system with only one processor, you get a warning message.

```
% setenv OMP_NUM_THREADS 2
% a.out
Warning: OMP_NUM_THREADS or NCPUS (2) greater
than available cpus (1)
FORTRAN STOP
```

Setting MP_WARN to NO eliminates these warning messages.

NCPUS

You can use the NCPUS environment variable to set the number of processes or threads used in parallel regions. The default is to use only one process or thread, which is known as serial mode.

Note

OMP_NUM_THREADS has the same functionality as NCPUS. For historical reasons, PGI supports the environment variable NCPUS. If both OMP_NUM_THREADS and NCPUS are set, the value of OMP_NUM_THREADS takes precedence.

Warning

Setting NCPUS to a value larger than the number of physical processors or cores in your system can cause parallel programs to run very slowly.

NCPUS_MAX

You can use the NCPUS_MAX environment variable to limit the maximum number of processes or threads used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using set_omp_num_threads(), will cause the number of processes or threads to be set at the value of NCPUS_MAX rather than the value specified in the function call.

NO_STOP_MESSAGE

If the NO_STOP_MESSAGE variable exists, the execution of a plain STOP statement does not produce the message FORTRAN STOP. The default behavior of the PGI Fortran compilers is to issue this message.

PATH

The PATH variable sets the directories that are searched for commands that the user types. When using PGI products, it is important that you set your PATH to include the location of the PGI products.

You can also use this variable to specify that you want to use only the linux86 version of the compilers and tools, or to target linux86 as the default.

The following csh example targets x64 linux86-64 version of the compilers and tools.

```
% set path = (/opt/pgi/linux86-64/13.10/bin $path)
```

Important

PGI

The PGI environment variable specifies the root directory where the PGI compilers and tools are installed. This variable is recognized at compile-time. If it is not set, the default value depends on your system as well as which compilers are installed:

- On Linux, the default value of this variable is /opt/pgi.
- On Windows, the default value is C:\Program Files\PGI, where C represents the system drive. If both 32- and 64-bit compilers are installed, the 32-bit compilers are in C:\Program Files (x86)\PGI .

- On Mac OS X, the default value of this variable is `/opt/pgi`.

In most cases, if the PGI environment variable is not set, the PGI compilers and tools dynamically determine the location of this root directory based on the instance of the compiler or tool that was invoked. However, there are still some dependencies on the PGI environment variable, and you can use it as a convenience when initializing your environment for use of the PGI compilers and tools.

For example, assuming you use csh and want the 64-bit linux86-64 versions of the PGI compilers and tools to be the default, you would use this syntax:

```
% setenv PGI /usr/pgi
% setenv MANPATH "$MANPATH":$PGI/linux86/13.10/man
% setenv LM_LICENSE_FILE $PGI/license.dat
% set path = ($PGI/linux86-64/13.10/bin $path)
```

PGI_CONTINUE

You set the `PGI_CONTINUE` variable to specify the actions to take before continuing with execution. For example, if the `PGI_CONTINUE` environment variable is set and then a program that is compiled with `-Mchkfpstk` is executed, the stack is automatically cleaned up and execution then continues. If `PGI_CONTINUE` is set to `verbose`, the stack is automatically cleaned up, a warning message is printed, and then execution continues.

Note

There is a performance penalty associated with the stack cleanup.

PGI_OBJSUFFIX

You can set the `PGI_OBJSUFFIX` environment variable to generate object files that have a specific suffix. For example, if you set `PGI_OBJSUFFIX` to `.o`, the object files have a suffix of `.o` rather than `.obj`.

PGI_STACK_USAGE

(Windows only) The `PGI_STACK_USAGE` variable allows you to explicitly set stack properties for your program. When the user compiles a program with the `-Mchkstk` option and sets the PGI_STACK_USAGE environment variable to any value, the program displays the stack space allocated and used after the program exits. You might see something similar to the following message:

```
thread 0 stack: max 8180KB, used 48KB
```

This message indicates that the program used 48KB of a 8180KB allocated stack. For more information on the `-Mchkstk` option, refer to "–Mchkstk" in the PGI Compiler Reference Manual.

PGI_TERM

The `PGI_TERM` environment variable controls the stack traceback and just-in-time debugging functionality. The runtime libraries use the value of `PGI_TERM` to determine what action to take when a program abnormally terminates.

The value of `PGI_TERM` is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:

```
% setenv PGI_TERM option[,option...]
```

- In bash, sh, zsh, or ksh:

```
$ PGI_TERM=option[,option...]
$ export PGI_TERM
```

- In the Windows Command Prompt:

```
C:\> set PGI_TERM=option[,option...]
```

Table 11.2 lists the supported values for `option`. Following the table is a complete description of each option that indicates specifically how you might apply the option.

By default, all of these options are disabled.

Table 11.2. Supported PGI_TERM Values

[no]debug	Enables/disables just-in-time debugging (debugging invoked on error)
[no]trace	Enables/disables stack traceback on error
[no]signal	Enables/disables establishment of signal handlers for common signals that cause program termination
[no]abort	Enables/disables calling the system termination routine abort()

[no]debug

This enables/disables just-in-time debugging. The default is `nodebug`.

When `PGI_TERM` is set to `debug`, the following command is invoked on error, unless you use `PGI_TERM_DEBUG` to override this default.

```
pgdbg -text -attach <pid>
```

`<pid>` is the process ID of the process being debugged.

The PGI_TERM_DEBUG environment variable may be set to override the default setting. For more information, refer to "PGI_TERM_DEBUG," on page 145.

[no]trace

[no]signal

This enables/disables establishing signal handlers for the most common signals that cause program termination. The default is `nosignal`. Setting `trace` and `debug` automatically enables `signal`. Specifically setting `nosignal` allows you to override this behavior.

[no]abort

This enables/disables calling the system termination routine abort(). The default is `noabort`. When `noabort` is in effect the process terminates by calling `_exit(127)`.

On Linux, when `abort` is in effect, the abort routine creates a core file and exits with code 127.

On Windows, when `abort` is in effect, the abort routine exits with the status of the exception received. For example, if the program receives an access violation, abort() exits with status 0xC0000005.

A few runtime errors just print an error message and call `exit(127)`, regardless of the status of `PGI_TERM`. These are mainly errors such as specifying an invalid environment variable value where a traceback would not be useful.

If it appears that abort() does not generate core files on a Linux system, be sure to unlimit the coredumpsize. You can do this in these ways:

- Using csh:

```
% limit coredumpsize unlimited
% setenv PGI_TERM abort
```

- Using bash, sh, zsh, or ksh:

```
$ ulimit -c unlimited
$ export PGI_TERM=abort
```

To debug a core file with pgdbg, start pgdbg with the -core option. For example, to view a core file named "core" for a program named "a.out":

```
$ pgdbg -core core a.out
```

For more information on why to use this variable, refer to "Stack Traceback and JIT Debugging," on page 147.

PGI_TERM_DEBUG

The `PGI_TERM_DEBUG` variable may be set to override the default behavior when `PGI_TERM` is set to `debug`.

The value of `PGI_TERM_DEBUG` should be set to the command line used to invoke the program. For example:

```
gdb --quiet --pid %d
```

The first occurrence of `%d` in the `PGI_TERM_DEBUG` string is replaced by the process id. The program named in the `PGI_TERM_DEBUG` string must be found on the current `PATH` or specified with a full path name.

PWD

The PWD variable allows you to display the current directory.

STATIC_RANDOM_SEED

You can use `STATIC_RANDOM_SEED` to force the seed returned by the Fortran 90/95 `RANDOM_SEED` intrinsic to be constant. The first call to `RANDOM_SEED` without arguments resets the random seed to a default value, then advances the seed by a variable amount based on time. Subsequent calls to `RANDOM_SEED` without arguments reset the random seed to the same initial value as the first call. Unless the time is exactly the same, each time a program is run a different random number sequence is generated. Setting the environment variable `STATIC_RANDOM_SEED` to `YES` forces the seed returned by `RANDOM_SEED` to be constant, thereby generating the same sequence of random numbers at each execution of the program.

TMP

You can use TMP to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools. This variable is interchangeable with TMPDIR.

TMPDIR

You can use TMPDIR to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools.

Using Environment Modules on Linux

On Linux, if you use the Environment Modules package, that is, the module load command, PGI includes a script to set up the appropriate module files.

Assuming your installation base directory is /opt/pgi, and your MODULEPATH environment variable is /usr/local/Modules/modulefiles, execute this command:

```
% /opt/pgi/linux86/13.10-0/etc/modulefiles/pgi.module.install \
 -all -install /usr/local/Modules/modulefiles
```

This command creates module files for all installed versions of the PGI compilers. You must have write permission to the modulefiles directory to enable the module commands:

```
% module load pgi32/13.10
% module load pgi64/13.10
% module load pgi/13.10
```

where "pgi/13.10" uses the 32-bit compilers on a 32-bit system and uses 64-bit compilers on a 64-bit system.

To see what versions are available, use this command:

```
% module avail pgi
```

The module load command sets or modifies the environment variables as indicated in the following table.

This Environment Variable...	Is set or modified by the module load command to ...
CC	Full path to pgcc
CPP	Full path to pgCC
CXX	Path to pgCC
C++	Path to pgCC
FC	Full path to pgfortran
F77	Full path to pgf77
F90	Full path to pgf90
LD_LIBRARY_PATH	Prepends the PGI library directory
MANPATH	Prepends the PGI man page directory
PATH	Prepends the PGI compiler and tools bin directory

This Environment Variable...	Is set or modified by the module load command to ...
PGI	The base installation directory
V	Full path to pgCC

Note

PGI does not provide support for the Environment Modules package. For more information about the package, go to: http://modules.sourceforge.net.

Stack Traceback and JIT Debugging

When a programming error results in a runtime error message or an application exception, a program will usually exit, perhaps with an error message. The PGI runtime library includes a mechanism to override this default action and instead print a stack traceback, start a debugger, or, on Linux, create a core file for post-mortem debugging.

The stack traceback and just-in-time debugging functionality is controlled by an environment variable, PGI_TERM, described in "PGI_TERM," on page 143. The runtime libraries use the value of PGI_TERM to determine what action to take when a program abnormally terminates.

When the PGI runtime library detects an error or catches a signal, it calls the routine pgi_stop_here() prior to generating a stack traceback or starting the debugger. The pgi_stop_here() routine is a convenient spot to set a breakpoint when debugging a program.

Chapter 12. Distributing Files - Deployment

Once you have successfully built, debugged and tuned your application, you may want to distribute it to users who need to run it on a variety of systems. This chapter addresses how to effectively distribute applications built using PGI compilers and tools. The application must be installed in such a way that it executes accurately on a system other than the one on which it was built, and which may be configured differently.

Deploying Applications on Linux

To successfully deploy your application on Linux, there are a number of issues to consider, including these:

- Runtime Libraries
- 64-bit Linux Systems
- Redistribution of Files
- Licensing

Runtime Library Considerations

On Linux systems, the system runtime libraries can be linked to an application either statically, or dynamically, For example, for the C runtime library, libc, you can use either the static version `libc.a` or the shared object `libc.so`. If the application is intended to run on Linux systems other than the one on which it was built, it is generally safer to use the shared object version of the library. This approach ensures that the application uses a version of the library that is compatible with the system on which the application is running. Further, it works best when the application is linked on a system that has an equivalent or earlier version of the system software than the system on which the application will be run.

Note

Building on a newer system and running the application on an older system may not produce the desired output.

To use the shared object version of a library, the application must also link to shared object versions of the PGI runtime libraries. To execute an application built in such a way on a system on which PGI compilers are

not installed, those shared objects must be available.To build using the shared object versions of the runtime libraries, use the `-Bdynamic` option, as shown here:

```
$ pgf90 -Bdynamic myprog.f90
```

64-bit Linux Considerations

On 64-bit Linux systems, 64-bit applications that use the `-mcmodel=medium` option sometimes cannot be successfully linked statically. Therefore, users with executables built with the `-mcmodel=medium` option may need to use shared libraries, linking dynamically. Also, runtime libraries built using the `-fpic` option use 32-bit offsets, so they sometimes need to reside near other runtime `libs` in a shared area of Linux program memory.

Note

If your application is linked dynamically using shared objects, then the shared object versions of the PGI runtime are required.

Linux Redistributable Files

The method for installing the shared object versions of the runtime libraries required for applications built with PGI compilers and tools is manual distribution.

When the PGI compilers are installed, there are directories that have a name that begins with `REDIST` for each platform (linux86 and linux86-64); these directories contain the redistributed shared object libraries. These may be redistributed by licensed PGI customers under the terms of the PGI End-User License Agreement.

Restrictions on Linux Portability

You cannot expect to be able to run an executable on any given Linux machine. Portability depends on the system you build on as well as how much your program uses system routines that may have changed from Linux release to Linux release. For example, one area of significant change between some versions of Linux is in `libpthread.so`. PGI compilers use this shared object for both the option `-Mconcur` (auto-parallel) and the option `-mp` (OpenMP) programs.

Typically, portability is supported for forward execution, meaning running a program on the same or a later version of Linux; but not for backward compatibility, that is, running on a prior release. For example, a user who compiles and links a program under Suse 9.1 should not expect the program to run without incident on a Red Hat 9.0 system, which is an earlier version of Linux. It *may* run, but it is less likely. Developers might consider building applications on earlier Linux versions for wider usage.

Licensing for Redistributable Files

The files in the REDIST directories may be redistributed under the terms of the End-User License Agreement for the product in which they were included.

Deploying Applications on Windows

Windows programs may be linked statically or dynamically.

- A statically linked program is completely self-contained, created by linking to static versions of the PGI and Microsoft runtime libraries.
- A dynamically linked program depends on separate dynamically-linked libraries (DLLs) that must be installed on a system for the application to run on that system.

Although it may be simpler to install a statically linked executable, there are advantages to using the DLL versions of the runtime, including these:

- Executable binary file size is smaller.
- Multiple processes can use DLLs at once, saving system resources.
- New versions of the runtime can be installed and used by the application without rebuilding the application.

Dynamically-linked Windows programs built with PGI compilers depend on dynamic runtime library files (DLLs). These DLLs must be distributed with such programs to enable them to execute on systems where the PGI compilers are not installed. These redistributable libraries include both PGI runtime libraries and Microsoft runtime libraries.

PGI Redistributables

PGI redistributable directories contain all of the PGI Linux runtime library shared object files or Windows dynamically-linked libraries that can be re-distributed by PGI 13.10 licensees under the terms of the PGI End-user License Agreement (EULA).

Microsoft Redistributables

The PGI products on Windows include Microsoft Open Tools. The Microsoft Open Tools directory contains a subdirectory named `redist`. PGI licensees may redistribute the files contained in this directory in accordance with the terms of the PGI End-User License Agreement.

Microsoft supplies installation packages, `vcredist_x86.exe` and `vcredist_x64.exe`, containing these runtime files. These files are available in the `redist` directory.

Code Generation and Processor Architecture

The PGI compilers can generate much more efficient code if they know the specific x86 processor architecture on which the program will run. When preparing to deploy your application, you should determine whether you want the application to run on the widest possible set of x86 processors, or if you want to restrict the application to run on a specific processor or set of processors. The restricted approach allows you to optimize performance for that set of processors.

Different processors have differences, some subtle, in hardware features, such as instruction sets and cache size. The compilers make architecture-specific decisions about such things as instruction selection, instruction scheduling, and vectorization, all of which can have a profound effect on the performance of your application.

Processor-specific code generation is controlled by the `-tp` option, described in the section "–tp <target> [,target...]" of the PGI Compiler Reference Manual. When an application is compiled without any `-tp` options, the compiler generates code for the type of processor on which the compiler is run.

Generating Generic x86 Code

To generate generic x86 code, use one of the following forms of the `-tp` option on your command line:

```
-tp px ! generate code for any x86 cpu type
```

```
-tp p6 ! generate code for Pentium 2 or greater
```

While both of these examples are good choices for portable execution, most users have Pentium 2 or greater CPUs.

Generating Code for a Specific Processor

You can use the `-tp` option to request that the compiler generate code optimized for a specific processor. The PGI Release Notes contains a list of supported processors or you can look at the `-tp` entry in the compiler output generated by using the `-help` option, described in "–help" in the PGI Compiler Reference Manual..

Generating One Executable for Multiple Types of Processors

PGI unified binaries provide a low-overhead method for a single program to run well on a number of hardware platforms.

All 64-bit PGI compilers can produce PGI Unified Binary programs that contain code streams fully optimized and supported for both AMD64 and Intel EM64T processors using the `-tp` target option.

The compilers generate and combine multiple binary code streams into one executable, where each stream is optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of 10-90% compared to generating full copies of code for each target.

Programs can use PGI Unified Binary technology even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mpfi` option disables generation of PGI Unified Binary object files. Instead, the default target auto-detect rules for the host are used to select the target processor.

PGI Unified Binary Command-line Switches

The PGI Unified Binary command-line switch is an extension of the target processor switch, `-tp`, which may be applied to individual files during compilation .

The target processor switch, `-tp`, accepts a comma-separated list of 64-bit targets and generates code optimized for each listed target.

The following example generates optimized code for three targets:

```
-tp k8-64,p7-64,core2-64
```

A special target switch, `-tp x64`, is the same as `-tp k8-64, p7-64`.

PGI Unified Binary Directives and Pragmas

PGI Unified binary directives and pragmas may be applied to functions, subroutines, or whole files. The directives and pragmas cause the compiler to generate PGI Unified Binary code optimized for one or more targets. No special command line options are needed for these pragmas and directives to take effect.

The syntax of the Fortran directive is this:

```
pgi$[g|r| ] pgi tp [target]...
```

where the scope is g (global), r (routine) or blank. The default is r, routine.

For example, the following syntax indicates that the whole file, represented by g, should be optimized for both k8_64 and p7_64.

```
pgi$g pgi tp k8_64 p7_64
```

The syntax of the C/C++ pragma is this:

```
#pragma [global|routine|] tp [target]...
```

where the scope is global, routine, or blank. The default is routine.

For example, the following syntax indicates that the next function should be optimized for k8_64, p7_64, and core2_64.

```
#pragma routine tp k8_64 p7_64 core2_64
```

Chapter 13. Inter-language Calling

This chapter describes inter-language calling conventions for C, C++, and Fortran programs using the PGI compilers. The following sections describe how to call a Fortran function or subroutine from a C or C++ program and how to call a C or C++ function from a Fortran program. For information on calling assembly language programs, refer to the "Runtime Environment" chapter of the PGI Compiler Reference Manual.

This chapter provides examples that use the following options related to inter-language calling. For more information on these options, refer to the "Command-Line Options Reference" chapter of the PGI Compiler Reference Manual.

`-c` `-Mnomain` `-Miface` `-Mupcase`

Overview of Calling Conventions

This chapter includes information on the following topics:

- Functions and subroutines in Fortran, C, and C++
- Naming and case conversion conventions
- Compatible data types
- Argument passing and special return values
- Arrays and indexes
- Win32 calling conventions

The sections "Inter-language Calling Considerations," on page 156 through "Example - C++ Calling Fortran," on page 165 describe how to perform inter-language calling using the Linux, Mac OSX, or Win64 convention. Default Fortran calling conventions for Win32 differ, although Win32 programs compiled using the `-Miface=unix` Fortran command-line option use the Linux/Win64 convention rather than the default Win32 conventions. All information in those sections pertaining to compatibility of arguments applies to Win32 as well. For details on the symbol name and argument passing conventions used on Win32 platforms, refer to "Win32 Calling Conventions," on page 167.

Inter-language Calling Considerations

In general, when argument data types and function return values agree, you can call a C or C++ function from Fortran as well as call a Fortran function from C or C++. When data types for arguments do not agree, you may need to develop custom mechanisms to handle them. For example, the Fortran `COMPLEX` type has a matching type in C99 but does not have a matching type in C90; however, it is still possible to provide inter-language calls but there are no general calling conventions for such cases.

Note

- If a C++ function contains objects with constructors and destructors, calling such a function from either C or Fortran is not possible unless the initialization in the main program is performed from a C++ program in which constructors and destructors are properly initialized.
- In general, you can call a C or Fortran function from C++ without problems as long as you use the extern "C" keyword to declare the function in the C++ program. This declaration prevents name mangling for the C function name. If you want to call a C++ function from C or Fortran, you also have to use the extern "C" keyword to declare the C++ function. This keeps the C++ compiler from mangling the name of the function.
- You can use the __cplusplus macro to allow a program or header file to work for both C and C++. For example, the following defines in the header file stdio.h allow this file to work for both C and C++.

```
#ifndef _STDIO_H
#define _STDIO_H
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
. /* Functions and data types defined... */
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif
```

- C++ member functions cannot be declared `extern`, since their names will always be mangled. Therefore, C++ member functions cannot be called from C or Fortran.

Functions and Subroutines

Fortran, C, and C++ define functions and subroutines differently.

For a Fortran program calling a C or C++ function, observe the following return value convention:

- When a C or C++ function returns a value, call it from Fortran as a function.
- When a C or C++ function does not return a value, call it as a subroutine.

For a C/C++ program calling a Fortran function, the call should return a similar type. Table 13.1, “Fortran and C/C++ Data Type Compatibility” lists compatible types. If the call is to a Fortran subroutine, a Fortran `CHARACTER` function, or a Fortran `COMPLEX` function, call it from C/C++ as a function that returns void. The

exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C/C++ as a function returning `int` whose value is the value of the integer expression specified in the alternate `RETURN` statement.

Upper and Lower Case Conventions, Underscores

By default on Linux, Win64, and OSX systems, all Fortran symbol names are converted to lower case. C and C++ are case sensitive, so upper-case function names stay upper-case. When you use inter-language calling, you can either name your C/C++ functions with lower-case names, or invoke the Fortran compiler command with the option `-Mupcase`, in which case it will not convert symbol names to lower-case.

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, and OSX systems, an underscore is appended to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C/C++ name space. Use these naming conventions:

- If you call a C/C++ function from Fortran, you should rename the C/C++ function by appending an underscore or use C$PRAGMA C in the Fortran program. For more information on `C$PRAGMA C`, refer to "C$PRAGMA C," on page 115.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

Compatible Data Types

Table 13.1 shows compatible data types between Fortran and C/C++. Table 13.2, "Fortran and C/C++ Representation of the `COMPLEX` Type," on page 158 shows how the Fortran `COMPLEX` type may be represented in C/C++.

Tip

If you can make your function/subroutine parameters as well as your return values match types, you should be able to use inter-language calling.

Table 13.1. Fortran and C/C++ Data Type Compatibility

Fortran Type (lower case)	C/C++ Type	Size (bytes)
character x	char x	1
character*n x	char x[n]	n
real x	float x	4
real*4 x	float x	4
real*8 x	double x	8
double precision	double x	8
integer x	int x	4
integer*1 x	signed char x	1
integer*2 x	short x	2

Fortran Type (lower case)	C/C++ Type	Size (bytes)
integer*4 x	int x	4
integer*8 x	long long x	8
logical x	int x	4
logical*1 x	char x	1
logical*2 x	short x	2
logical*4	int x	4
logical*8	long x	8

Table 13.2. Fortran and C/C++ Representation of the **COMPLEX** Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8 8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

Note

For C/C++, the `complex` type implies C99 or later.

Fortran Named Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore. For example, here is a Fortran common block:

```
INTEGER I
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, c, cd, d
```

This Fortran Common Block is represented in C with the following equivalent:

```
extern struct {
 int i;
 struct {float real, imag;} c;
 struct {double real, imag;} cd;
 double d;
} com_;
```

This same Fortran Common Block is represented in C++ with the following equivalent:

```
extern "C" struct {
 int i;
 struct {float real, imag;} c;
 struct {double real, imag;} cd;
 double d;
} com_;
```

Tip

For global or external data sharing, `extern "C"` is not required.

Argument Passing and Return Values

In Fortran, arguments are passed by reference, that is, the address of the argument is passed, rather than the argument itself. In C/C++, arguments are passed by value, except for strings and arrays, which are passed by reference. Due to the flexibility provided in C/C++, you can work around these differences. Solving the parameter passing differences generally involves intelligent use of the `&` and `*` operators in argument passing when C/C++ calls Fortran and in argument declarations when Fortran calls C/C++.

For strings declared in Fortran as type `CHARACTER`, an argument representing the length of the string is also passed to a calling function.

On Linux and Mac OS X systems, or when using the UNIX calling convention on Windows (option `-Miface=unix`), the compiler places the length argument(s) at the end of the parameter list, following the other formal arguments. The length argument is passed by value, not by reference.

Passing by Value (%VAL)

When passing parameters from a Fortran subprogram to a C/C++ function, it is possible to pass by value using the `%VAL` function. If you enclose a Fortran parameter with `%VAL()`, the parameter is passed by value. For example, the following call passes the integer `i` and the logical `bvar` by value.

```
integer*1 i
logical*1 bvar
call cvalue (%VAL(i), %VAL(bvar))
```

Character Return Values

"Functions and Subroutines," on page 156 describes the general rules for return values for C/C++ and Fortran inter-language calling. There is a special return value to consider. When a Fortran function returns a character, two arguments need to be added at the beginning of the C/C++ calling function's argument list:

- The address of the return character or characters
- The length of the return character

Example 13.1, "Character Return Parameters" illustrates the extra parameters, `tmp` and `10`, supplied by the caller:

Example 13.1. Character Return Parameters

```
! Fortran function returns a character
CHARACTER*(*) FUNCTION CHF(C1,I)
   CHARACTER*(*) C1
   INTEGER I
END
```

```
/* C declaration of Fortran function */
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

If the Fortran function is declared to return a character value of constant length, for example `CHARACTER*4 FUNCTION CHF()`, the second extra parameter representing the length must still be supplied, but is not used.

Note

The value of the character function is not automatically NULL-terminated.

Complex Return Values

When a Fortran function returns a complex value, an argument needs to be added at the beginning of the C/C++ calling function's argument list; this argument is the address of the complex return value. Example 13.2, "COMPLEX Return Values" illustrates the extra parameter, `cplx`, supplied by the caller.

Example 13.2. COMPLEX Return Values

```
COMPLEX FUNCTION CF(C, I)
   INTEGER I
 . . .
END
```

```
extern void cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int i;
cf_(&c1, &i);
```

Array Indices

C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. If you adjust your array comparisons so that a Fortran second element is compared to a C/C++ first element, and adjust similarly for other elements, you should not have problems working with this difference. If this is not satisfactory, you can declare your Fortran arrays to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ uses row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. For arrays other than single dimensional arrays, and square two-dimensional arrays, inter-language function mixing is not recommended.

Examples

This section contains examples that illustrate inter-language calling.

Example - Fortran Calling C

Note

There are other solutions to calling C from Fortran than the one presented in this section. For example, you can use the `iso_c_binding` intrinsic module which PGI does support. For more information on this module and for examples of how to use it, search the web using the keyword iso_c_binding.

Example 13.4, "C function f2c_func_" shows a C function that is called by the Fortran main program shown in Example 13.3, "Fortran Main Program f2c_main.f". Notice that each argument is defined as a pointer, since Fortran passes by reference. Also notice that the C function name uses all lower-case and a trailing "_".

Example 13.3. Fortran Main Program f2c_main.f

```
logical*1 bool1
 character letter1
 integer*4 numint1, numint2
 real numfloat1
 double precision numdoub1
 integer*2 numshor1
 external f2c_func

 call f2c_func(bool1, letter1, numint1, numint2, numfloat1, numdoub1, numshor1)

 write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
 + bool1, letter1, numint1, numint2, numfloat1,numdoub1, numshor1

 end
```

Example 13.4. C function f2c_func_

```
#define TRUE 0xff
#define FALSE 0
void f2c_func_( bool1, letter1, numint1, numint2, numfloat1,\
 numdoub1, numshor1, len_letter1)
 char *bool1, *letter1;
 int *numint1, *numint2;
 float *numfloat1;
 double *numdoub1;
 short *numshor1;
 int len_letter1;
{
 *bool1 = TRUE;  *letter1 = 'v';
 *numint1 = 11;  *numint2 = -44;
 *numfloat1 = 39.6 ;
 *numdoub1 = 39.2;
 *numshor1 = 981;
}
```

Compile and execute the program `f2c_main.f` with the call to `f2c_func_` using the following command lines:

```
$ pgcc -c f2c_func.c
$ pgfortran f2c_func.o f2c_main.f
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C Calling Fortran

Example 13.5, "C Main Program c2f_main.c" shows a C main program that calls the Fortran subroutine shown in Example 13.6, "Fortran Subroutine c2f_sub.f".

- Each call uses the & operator to pass by reference.
- The call to the Fortran subroutine uses all lower-case and a trailing "_".

Example 13.5. C Main Program c2f_main.c

```
void main () {
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;
 extern void c2f_func_();
 c2f_sub_(&bool1,&letter1,&numint1,&numint2,&numfloat1,&numdoub1,&numshor1, 1);
 printf(" %s %c %d %d %3.1f %.0f %d\n",
 bool1?"TRUE":"FALSE", letter1, numint1, numint2,
 numfloat1, numdoub1, numshor1);
}
```

Example 13.6. Fortran Subroutine c2f_sub.f

```
subroutine c2f_func ( bool1, letter1, numint1, numint2,
+ numfloat1, numdoub1, numshor1)
   logical*1 bool1
   character letter1
   integer numint1, numint2
   double precision numdoub1
   real numfloat1
   integer*2 numshor1

   bool1 = .true.
   letter1 = "v"
    numint1 = 11
    numint2 = -44
    numdoub1 = 902
    numfloat1 = 39.6
    numshor1 = 299
    return
end
```

To compile this Fortran subroutine and C program, use the following commands:

```
$ pgcc -c c2f_main.c
$ pgfortran -Mnomain c2f_main.o c2_sub.f
```

Executing the resulting `a.out` file should produce the following output:

```
TRUE v 11 -44 39.6 902 299
```

Example - C++ Calling C

Example 13.7, "C++ Main Program cp2c_main.C Calling a C Function" shows a C++ main program that calls the C function shown in Example 13.8, "Simple C Function c2cp_func.c".

Example 13.7. C++ Main Program cp2c_main.C Calling a C Function

```
extern "C" void cp2c_func(int n, int m, int *p);
#include <iostream>
main()
{
 int a,b,c;
 a=8;
 b=2;
 c=0;
 cout << "main: a = "<<a<<" b = "<<b<<"ptr c = "<<hex<<&c<< endl;
 cp2c_func(a,b,&c);
 cout << "main: res = "<<c<<endl;
}
```

Example 13.8. Simple C Function c2cp_func.c

```
void cp2c_func(num1, num2, res)
int num1, num2, *res;
{
 printf("func: a = %d b = %d ptr c = %x\n",num1,num2,res);
 *res=num1/num2;
 printf("func: res = %d\n",*res);
}
```

To compile this C function and C++ main program, use the following commands:

```
$ pgcc -c cp2c_func.c
$ pgcpp cp2c_main.C cp2c_func.o
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```

Example - C Calling C++

Example 13.9, "C Main Program c2cp_main.c Calling a C++ Function" shows a C main program that calls the C++ function shown in Example 13.10, "Simple C++ Function c2cp_func.C with Extern C".

Example 13.9. C Main Program c2cp_main.c Calling a C++ Function

```
extern void c2cp_func(int a, int b, int *c);
#include <stdio.h>
main() {
 int a,b,c;
 a=8; b=2;
 printf("main: a = %d b = %d ptr c = %x\n",a,b,&c);
 c2cp_func(a,b,&c);
 printf("main: res = %d\n",c);
}
```

Example 13.10. Simple C++ Function c2cp_func.C with Extern C

```
#include <iostream>
extern "C" void c2cp_func(int num1,int num2,int *res)
{
 cout << "func: a = "<<num1<<" b = "<<num2<<"ptr c ="<<res<<endl;
 *res=num1/num2;
 cout << "func: res = "<<res<<endl;
}
```

To compile this C function and C++ main program, use the following commands:

```
$ pgcc -c c2cp_main.c
$ pgcpp c2cp_main.o c2cp_func.C
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```

Note

You cannot use the extern "C" form of declaration for an object's member functions.

Example - Fortran Calling C++

The Fortran main program shown in Example 13.11, "Fortran Main Program f2cp_main.f calling a C++ function" calls the C++ function shown in Example 13.12, "C++ function f2cp_func.C".

Notice:

- Each argument is defined as a pointer in the C++ function, since Fortran passes by reference.
- The C++ function name uses all lower-case and a trailing "_":

Example 13.11. Fortran Main Program f2cp_main.f calling a C++ function

```
  logical*1 bool1
   character letter1
   integer*4 numint1, numint2
   real numfloat1
   double precision numdoub1
   integer*2 numshor1
 external f2cpfunc
 call f2cp_func (bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
   write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
   + bool1, letter1, numint1, numint2, numfloat1,
   + numdoub1, numshor1
 end
```

Example 13.12. C++ function f2cp_func.C

```
#define TRUE 0xff
#define FALSE 0
extern "C"
{
extern void f2cp_func_ (
 char *bool1, *letter1,
 int *numint1, *numint2,
 float *numfloat1,
 double *numdoub1,
 short *numshort1,
 int len_letter1)
{
 *bool1 = TRUE;     *letter1 = 'v';
 *numint1 = 11;     *numint2 = -44;
 *numfloat1 = 39.6; *numdoub1 = 39.2;   *numshort1 = 981;
}
}
```

Assuming the Fortran program is in a file `fmain.f`, and the C++ function is in a file `cpfunc.C`, create an executable, using the following command lines:

```
$ pgcpp -c f2cp_func.C
$ pgfortran f2cp_func.o f2cp_main.f -pgcpplibs
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C++ Calling Fortran

Example 13.14, "Fortran Subroutine cp2f_func.f" shows a Fortran subroutine called by the C++ main program shown in Example 13.13, "C++ main program cp2f_main.C". Notice that each call uses the & operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing "_":

Example 13.13. C++ main program cp2f_main.C

```
#include <iostream>
extern "C" { extern void cp2f_func_(char *,char *,int *,int *,
 float *,double *,short *); }
main ()
{
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;

 cp2f_func(&bool1,&letter1,&numint1,&numint2,&numfloat1, &numdoub1,&numshor1);
 cout << " bool1 = ";
 bool1?cout << "TRUE ":cout << "FALSE "; cout <<endl;
 cout << " letter1 = " << letter1 <<endl;
 cout << " numint1 = " << numint1 <<endl;
 cout << " numint2 = " << numint2 <<endl;
 cout << " numfloat1 = " << numfloat1 <<endl;
 cout << " numdoub1 = " << numdoub1 <<endl;
 cout << " numshor1 = " << numshor1 <<endl;
}
```

Example 13.14. Fortran Subroutine cp2f_func.f

```
 subroutine cp2f_func ( bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
 logical*1 bool1
 character letter1
 integer numint1, numint2
 double precision numdoub1
 real numfloat1
 integer*2 numshor1
 bool1 = .true. ; letter1 = "v"
 numint1 = 11 ; numint2 = -44
 numdoub1 = 902 ; numfloat1 = 39.6 ; numshor1 = 299
 return
 end
```

To compile this Fortran subroutine and C++ program, use the following command lines:

```
$ pgfortran -c cp2f_func.f
$ pgcpp cp2f_func.o cp2f_main.C -pgf90libs
```

Executing this C++ main should produce the following output:

```
bool1 = TRUE
letter1 = v
numint1 = 11
numint2 = -44
numfloat1 = 39.6
numdoub1 = 902
numshor1 = 299
```

Note that you must explicitly link in the PGFORTRAN runtime support libraries when linking pgfortran-compiled program units into C or C++ main programs. When linking pgf77-compiled program units into C or C++ main programs, you need only link in `-lpgftnrtl`.

Win32 Calling Conventions

A calling convention is a set of conventions that describe the manner in which a particular routine is executed. A routine's calling conventions specify where parameters and function results are passed. For a stack-based routine, the calling conventions determine the structure of the routine's stack frame.

The calling convention for C/C++ is identical between most compilers on Win32 , Linux, Mac OS X, and Win64. However, Fortran calling conventions vary widely between legacy Win32 Fortran compilers and Linux or Win64 Fortran compilers.

Win32 Fortran Calling Conventions

Four styles of calling conventions are supported using the PGI Fortran compilers for Win32: Default, C, STDCALL, and UNIX.

- **Default** - Used in the absence of compilation flags or directives to alter the default.
- **C or STDCALL** - Used if an appropriate compiler directive is placed in a program unit containing the call. The C and STDCALL conventions are typically used to call routines coded in C or assembly language that depend on these conventions.
- **UNIX** - Used in any Fortran program unit compiled using the `-Miface=unix` (or `-Munix`) compilation flag.

The following table outlines each of these calling conventions.

Table 13.3. Calling Conventions Supported by the PGI Fortran Compilers

Convention	Default	STDCALL	C	UNIX
Case of symbol name	Upper	Lower	Lower	Lower
Leading underscore	Yes	Yes	Yes	Yes
Trailing underscore	No	No	No	Yes
Argument byte count added	Yes	Yes	No	No
Arguments passed by reference	Yes	No*	No*	Yes
Character argument length passed	After each char argument	No	No	End of argument list
First character of character string is passed by value	No	Yes	Yes	No
varargs support	No	No	Yes	Yes
Caller cleans stack	No	No	Yes	Yes

* Except arrays, which are always passed by reference even in the STDCALL and C conventions

Note

While it is compatible with the Fortran implementations of Microsoft and several other vendors, the C calling convention supported by the PGI Fortran compilers for Windows is not strictly compatible with the C calling convention used by most C/C++ compilers. In particular, symbol names produced by PGI Fortran compilers using the C convention are all lower case. The standard C convention is to preserve mixed-case symbol names. You can cause any of the PGI Fortran compilers to preserve mixed-case symbol names using the `-Mupcase` option, but be aware that this could have other ramifications on your program.

Symbol Name Construction and Calling Example

This section presents an example of the rules outlined in Table 13.3, "Calling Conventions Supported by the PGI Fortran Compilers," on page 167. In the pseudocode shown in the following examples, `%addr` refers to the address of a data item while `%val` refers to the value of that data item. Subroutine and function names are converted into symbol names according to the rules outlined in Table 13.3.

Consider the following subroutine call, where a is a double precision scalar, b is a real vector of size n, and n is an integer:

```
call work ( 'ERR', a, b, n)
```

- **Default** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all upper case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Byte counts for character arguments appear immediately following the corresponding argument in the argument list.

  The following example is pseudocode for the preceding subroutine call using Default conventions:

  ```
  call _WORK@20 (%addr('ERR'), 3, %addr(a), %addr(b), %addr(n))
  ```

- **STDCALL** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using STDCALL conventions:

  ```
  call _work@20 (%val('E'), %val(a), %addr(b), %val(n))
  ```

  Notice in this case that there are still 20 bytes in the argument list. However, rather than five 4-byte quantities as in the Default convention, there are three 4-byte quantities and one 8-byte quantity (the double precision value of a).

- **C** - The symbol name for the subroutine is constructed by pre-pending an underscore and converting to all lower case. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using C conventions:

  ```
  call _work (%val('E'), %val(a), %addr(b), %val(n))
  ```

- **UNIX** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an underscore. Byte counts for character strings appear in sequence

following the last argument in the argument list. The following is an example of the pseudocode for the work subroutine call using UNIX conventions:

```
call _work_ (%addr('ERR'), %addr(a), %addr(b), %addr(n),3)
```

Using the Default Calling Convention

The Default calling convention is used if no directives are inserted to modify calling conventions and if neither the `-Miface=unix` (or `-Munix`) compilation flag is used. Refer to "Symbol Name Construction and Calling Example," on page 168 for a complete description of the Default calling convention.

Using the STDCALL Calling Convention

Using the STDCALL calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the STDCALL program unit. You cannot mix UNIX-style argument passing and STDCALL calling conventions within the same file.

In the following example syntax for the directive, `work` is the name of the subroutine to be called using STDCALL conventions:

```
!DEC$ ATTRIBUTES STDCALL :: work
```

You can list more than one subroutine, separating them by commas. Refer to "Symbol Name Construction and Calling Example," on page 168 for a complete description of the implementation of STDCALL.

Note

- The directive prefix !DEC$ requires a space between the prefix and the directive keyword ATTRIBUTES.
- The ! must begin the prefix when compiling using Fortran 90 freeform format.
- The characters C or * can be used in place of ! in either form of the prefix when compiling used fixed-form format.
- The directives are completely case insensitive.

Using the C Calling Convention

Using the C calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the C program unit. You cannot mix UNIX-style argument passing and C calling conventions within the same file.

Syntax for the directive is as follows:

```
!DEC$ ATTRIBUTES C :: work
```

Where `work` is the name of the subroutine to be called using C conventions. More than one subroutine may be listed, separated by commas. Refer to "Symbol Name Construction and Calling Example," on page 168 for a complete description of the implementation of the C calling convention.

Using the UNIX Calling Convention

Using the UNIX calling convention is straightforward. Any program unit compiled using `-Miface=unix` or the `-Munix` compilation flag uses the UNIX convention.

Using the CREF Calling Convention

Using the CREF calling convention is straightforward. Any program unit compiled using `-Miface=cref` compilation flag uses the CREF convention.

Chapter 14. Programming Considerations for 64-Bit Environments

PGI provides 64-bit compilers for the 64-bit Linux, Windows, and Mac OS X operating systems running on the x64 architecture. You can use these compilers to create programs that use 64-bit memory addresses. However, there are limitations to how this capability can be applied. With the exception of Linux86-64, the object file formats on all of the operating systems limit the total cumulative size of code plus static data to 2GB. This limit includes the code and statically declared data in the program and in system and user object libraries. Linux86-64 implements a mechanism that overcomes this limitations, as described in "Large Static Data in Linux," on page 172. This chapter describes the specifics of how to use the PGI compilers to make use of 64-bit memory addressing.

The 64-bit Windows, Linux, and Mac OS X environments maintain 32-bit compatibility, which means that 32-bit applications can be developed and executed on the corresponding 64-bit operating system.

Note

The 64-bit PGI compilers are 64-bit applications which cannot run on anything but 64-bit CPUs running 64-bit Operating Systems.

This chapter describes how to use the following options related to 64-bit programming.

`-fPIC`	`-mcmodel=medium`	`-Mlarge_arrays`
`-i8`	`-Mlargeaddressaware`	`-tp`

Data Types in the 64-Bit Environment

The size of some data types can be different in a 64-bit environment. This section describes the major differences. For detailed information, refer to the "Fortran, C, and C++ Data Types" chapter of the PGI Compiler Reference Manual.

C/C++ Data Types

On 32-bit Windows, int is 4 bytes, long is 4 bytes, and pointers are 4 bytes. On 64-bit windows, the size of an int is 4 bytes, a long is 4 bytes, and a pointer is 8 bytes.

On the 32-bit Linux and Mac OS X operating systems, the size of an int is 4 bytes, a long is 4 bytes, and a pointer is 4 bytes. On the 64-bit Linux and Mac OS X operating systems, the size of an int is 4 bytes, a long is 8 bytes, and a pointer is 8 bytes.

Fortran Data Types

In Fortran, the default size of the INTEGER type is 4 bytes. The `-i8` compiler option may be used to make the default size of all INTEGER data in the program 8 bytes.

When using the `-Mlarge_arrays` option, described in "64-Bit Array Indexing," on page 172, any 4-byte INTEGER variables that are used to index arrays are silently promoted by the compiler to 8 bytes. This promotion can lead to unexpected consequences, so 8-byte INTEGER variables are recommended for array indexing when using the option `-Mlarge_arrays`.

Large Static Data in Linux

Linux86-64 operating systems support two different memory models. The default model used by PGI compilers is the small memory model, which can be specified using -mcmodel=small. This is the 32-bit model, which limits the size of code plus statically allocated data, including system and user libraries, to 2GB. The medium memory model, specified by -mcmodel=medium, allows combined code and static data areas (.text and .bss sections) larger than 2GB. The `-mcmodel=medium` option must be used on both the compile command and the link command in order to take effect.

The Win64 and 64-bit Mac OS X operating systems do not have any support for large static data declarations.

There are two drawbacks to using `-mcmodel=medium`. First, there is increased addressing overhead to support the large data range. This can affect performance, though the compilers seek to minimize the added overhead through careful instruction generation. Second, -mcmodel=medium cannot be used for objects in shared libraries, because there is no OS support for 64-bit dynamic linkage.

Large Dynamically Allocated Data

Dynamically allocated data objects in programs compiled by the 64-bit PGI compilers can be larger than 2GB. No special compiler options are required to enable this functionality. The size of the allocation is only limited by the system. However, to correctly access dynamically allocated arrays with more than 2G elements you should use the `-Mlarge_arrays` option, described in the following section.

64-Bit Array Indexing

The 64-bit PGI compilers provide an option, `-Mlarge_arrays`, that enables 64-bit indexing of arrays. This means that, as necessary, 64-bit INTEGER constants and variables are used to index arrays.

Note

In the presence of `-Mlarge_arrays`, the compiler may silently promote 32-bit integers to 64 bits, which can have unexpected side effects.

On Linux86-64, the `-Mlarge_arrays` option also enables single static data objects larger than 2 GB. This option is the default in the presence of `-mcmodel=medium`.

Note

On Win64, static data may not be larger than 2GB.

Compiler Options for 64-bit Programming

The usual switches that apply to 64-bit programmers seeking to increase the data range of their applications are in the following table.

Table 14.1. 64-bit Compiler Options

Option	Purpose	Considerations
–mcmodel=medium	Enlarge object size; Allow for declared data the size of larger than 2GB	Linux86-64 only. Slower execution. Cannot be used with –fPIC. Objects cannot be put into shared libraries.
–Mlargeaddressaware	[Win64 only] Generates code that allows for addresses greater than 2GB, using RIP-relative addressing.	Use –Mlargeaddressaware=no for a direct addressing mechanism that restricts the total addressable memory. This is not applicable if the object file is placed in a DLL. Further, if an object file is compiled with this option, it must also be used when linking.
–Mlarge_arrays	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Is implicit with –mcmodel=medium. Can be used with option –mcmodel=small. Win64 does not support –Mlarge_arrays for static objects larger than 2GB.
–fPIC	Position independent code. Necessary for shared libraries.	Dynamic linking restricted to a 32-bit offset. External symbol references should refer to other shared lib routines, rather than the program calling them.
–i8	All INTEGER functions, data, and constants not explicitly declared INTEGER*4 are assumed to be INTEGER*8.	Users should take care to explicitly declare INTEGER functions as INTEGER*4.

The following table summarizes the limits of these programming models:

Table 14.2. Effects of Options on Memory and Array Sizes

Combined Compiler Options	Addr. Math		Max Size Gbytes			Comments
	A	I	AS	DS	TS	
`-tp k8-32` or `-tp p7`	32	32	2	2	2	32-bit linux86 programs
`-tp k8-64` or `-tp p7-64`	64	32	2	2	2	64-bit addr limited by option `-mcmodel=small`
`-tp k8-64 -fpic` or `-tp p7-64 -fpic`	64	32	2	2	2	`-fpic` *incompatible with* –mcmodel=medium
`-tp k8-64` or `-tp p7-64 -mcmodel=medium`	64	64	>2	>2	>2	Enable full support for 64-bit data addressing

Column Legend	
A	Address Type - size in bits of data used for address calculations, 32-bit or 64-bit.
I	Index Arithmetic - bit-size of data used to index into arrays and other aggregate data structures. If 32-bit, total range of any single data object is limited to 2GB.
AS	Maximum Array Size - the maximum size in gigabytes of any single data object.
DS	*Maximum Data Size* - max size in gigabytes combined of all data objects in .bss
TS	Maximum Total Size - max size in gigabytes, in aggregate, of all executable code and data objects in a running program.

Practical Limitations of Large Array Programming

The 64-bit addressing capability of the Linux86-64 and Win64 environments can cause unexpected issues when data sizes are enlarged significantly. The following table describes the most common occurrences of practical limitations of large array programming.

Table 14.3. 64-Bit Limitations

array initialization	Initializing a large array with a data statement may result in very large assembly and object files, where a line of assembler source is required for each element in the initialized array. Compilation and linking can be very time consuming as well. To avoid this issue, consider initializing large arrays in a loop at runtime rather than in a data statement.

stack space	Stack space can be a problem for data that is stack-based. In Win64, stack space can be increased by using this link-time switch, where N is the desired stack size:–`Wl,-stack:N`
	Note
	In linux86-64, stack size is increased in the environment. Setting stacksize to unlimited often is not large enough.
	`limit stacksize new_size ! in csh`
	`ulimit –s new_size ! in bash`

page swapping	If your executable is much larger than the physical size of memory, page swapping can cause it to run dramatically slower; it may even fail. This is not a compiler problem. Try smaller data sets to determine whether or not a problem is due to page thrashing.
configured space	Be sure your linux86-64 system is configured with swap space sufficiently large to support the data sets used in your application(s). If your memory+swap space is not sufficiently large, your application will likely encounter a segmentation fault at runtime.
support for large address offsets in object file format	Arrays that are not dynamically allocated are limited by how the compiler can express the 'distance' between them when generating code. A field in the object file stores this 'distance' value, which is limited to 32-bits on Win32, Win64, linux86, and linux86-64 with –mcmodel=small. It is 64-bits on linux86-64 with –mcmodel=medium. **Note** Without the 64-bit offset support in the object file format, large arrays cannot be declared statically or locally stack-based.

Medium Memory Model and Large Array in C

Consider the following example, where the aggregate size of the arrays exceeds 2GB.

Example 14.1. Medium Memory Model and Large Array in C

```
% cat bigadd.
#include <stdio.h>
#define SIZE 600000000 /* > 2GB/4 */
static float a[SIZE], b[SIZE];
int
main()
{
 long long i, n, m;
 float c[SIZE]; /* goes on stack */
 n = SIZE;
 m = 0;
 for (i = 0; i < n; i += 10000) {
 a[i] = i + 1;
 b[i] = 2.0 * (i + 1);
 c[i] = a[i] + b[i];
```

```
 m = i;
 }
 printf("a[0]=%g b[0]=%g c[0]=%g\n", a[0], b[0], c[0]);
 printf("m=%lld a[%lld]=%g b[%lld]=%gc[%lld]=%g\n",m,m,a[m],m,b[m],m,c[m]);
 return 0;
}
```

```
% pgcc –mcmodel=medium –o bigadd bigadd.c
```

When SIZE is greater than 2G/4, and the arrays are of type float with 4 bytes per element, the size of each array is greater than 2GB. With pgcc, using the –mcmodel=medium switch, a static data object can now be > 2GB in size. If you execute with these settings in your environment, you may see the following:

```
% bigadd
 Segmentation fault
```

Execution fails because the stack size is not large enough. You can most likely correct this error by using the **limit stacksize** command to reset the stack size in your environment:

```
% limit stacksize 3000M
```

Note

The command **limit stacksize unlimited** probably does not provide as large a stack as we are using in the Example 14.1.

```
% bigadd
a[0]=1 b[0]=2 c[0]=3
n=599990000 a[599990000]=5.9999e+08 b[599990000]=1.19998e+09
c[599990000]=1.79997e+09
```

The size of the bss section of the bigadd executable is now larger than 2GB:

```
% size –-format=sysv bigadd | grep bss
.bss 4800000008 5245696
% size –-format=sysv bigadd | grep Total
Total 4800005080
```

Medium Memory Model and Large Array in Fortran

The following example works with the PGFORTRAN, PGF95, and PGF77 compilers included in Release 2013. Both compilers use 64-bit addresses and index arithmetic when the –mcmodel=medium option is used.

Consider the following example:

Example 14.2. Medium Memory Model and Large Array in Fortran

```
% cat mat.f
 program mat
 integer i, j, k, size, l, m, n parameter (size=16000) ! >2GB
 parameter (m=size,n=size)
 real*8 a(m,n),b(m,n),c(m,n),d
 do i = 1, m
 do j = 1, n
 a(i,j)=10000.0D0*dble(i)+dble(j)
 b(i,j)=20000.0D0*dble(i)+dble(j)
 enddo
 enddo
```

```
!$omp parallel
!$omp do
 do i = 1, m
 do j = 1, n
 c(i,j) = a(i,j) + b(i,j)
 enddo
 enddo
!$omp do
 do i=1,m
 do j = 1, n
 d = 30000.0D0*dble(i)+dble(j)+dble(j)
 if(d .ne. c(i,j)) then
 print *,"err i=",i,"j=",j
 print *,"c(i,j)=",c(i,j)
 print *,"d=",d
 stop
 endif
 enddo
 enddo
!$omp end parallel
 print *, "M =",M,", N =",N
 print *, "c(M,N) = ", c(m,n)
 end
```

When compiled with the PGFORTRAN compiler using –mcmodel=medium:

```
% pgfortran –mp –o mat mat.f –i8 –mcmodel=medium
% setenv OMP_NUM_THREADS 2
% mat
M = 16000 , N = 16000
c(M,N) = 480032000.0000000
```

Large Array and Small Memory Model in Fortran

The following example uses large, dynamically-allocated arrays. The code is divided into a main and subroutine so you could put the subroutine into a shared library. Dynamic allocation of large arrays saves space in the size of executable and saves time initializing data. Further, the routines can be compiled with 32-bit compilers, by just decreasing the parameter size.

Example 14.3. Large Array and Small Memory Model in Fortran

```
% cat mat_allo.f90
program mat_allo
 integer i, j
 integer size, m, n
 parameter (size=16000)
 parameter (m=size,n=size)
 double precision, allocatable::a(:,:),b(:,:),c(:,:)
 allocate(a(m,n), b(m,n), c(m,n))
 do i = 100, m, 1
 do j = 100, n, 1
 a(i,j) = 10000.0D0 * dble(i) + dble(j)
 b(i,j) = 20000.0D0 * dble(i) + dble(j)
 enddo
 enddo
 call mat_add(a,b,c,m,n)
 print *, "M =",m,",N =",n
```

```
 print *, "c(M,N) = ", c(m,n)
end
subroutine mat_add(a,b,c,m,n)
 integer m, n, i, j
 double precision a(m,n),b(m,n),c(m,n)
!$omp do
 do i = 1, m
 do j = 1, n
 c(i,j) = a(i,j) + b(i,j)
 enddo
 enddo
 return
end
% pgfortran –o mat_allo mat_allo.f90 –i8 –Mlarge_arrays -mp -fast
```

Chapter 15. C/C++ Inline Assembly and Intrinsics

Inline Assembly

Inline Assembly lets you specify machine instructions inside a "C" function. The format for an inline assembly instruction is this:

```
{ asm | __asm__ } ("string");
```

The asm statement begins with the *asm* or *__asm__* keyword. The __asm__ keyword is typically used in header files that may be included in ISO "C" programs.

"*string*" is one or more machine specific instructions separated with a semi-colon (*;*) or newline (*\n*) character. These instructions are inserted directly into the compiler's assembly-language output for the enclosing function.

Some simple asm statements are:

```
asm ("cli");
asm ("sti");
```

These asm statements disable and enable system interrupts respectively.

In the following example, the eax register is set to zero.

```
asm( "pushl %eax\n\t" "movl $0, %eax\n\t" "popl %eax");
```

Notice that eax is pushed on the stack so that it is it not clobbered. When the statement is done with eax, it is restored with the popl instruction.

Typically a program uses macros that enclose asm statements. The following two examples use the interrupt constructs created previously in this section:

```
#define disableInt __asm__ ("cli");
#define enableInt __asm__ ("sti");
```

Extended Inline Assembly

"Inline Assembly," on page 179 explains how to use inline assembly to specify machine specific instructions inside a "C" function. This approach works well for simple machine operations such as disabling and enabling system interrupts. However, inline assembly has three distinct limitations:

1. The programmer must choose the registers required by the inline assembly.
2. To prevent register clobbering, the inline assembly must include push and pop code for registers that get modified by the inline assembly.
3. There is no easy way to access stack variables in an inline assembly statement.

Extended Inline Assembly was created to address these limitations. The format for extended inline assembly, also known as *extended asm*, is as follows:

```
{ asm | __asm__ } [ volatile | __volatile__ ]
("string" [: [output operands]] [: [input operands]] [: [clobberlist]]);
```

- Extended asm statements begin with the *asm* or *__asm__* keyword. Typically the *__asm__* keyword is used in header files that may be included by ISO "C" programs.
- An optional *volatile* or *__volatile__* keyword may appear after the *asm* keyword. This keyword instructs the compiler not to delete, move significantly, or combine with any other asm statement. Like __asm__, the __volatile__ keyword is typically used with header files that may be included by ISO "C" programs.
- "*string*" is one or more machine specific instructions separated with a semi-colon (*;*) or newline (*\n*) character. The string can also contain operands specified in the *[output operands]*, *[input operands]*, and *[clobber list]*. The instructions are inserted directly into the compiler's assembly-language output for the enclosing function.
- The *[output operands]*, *[input operands]*, and *[clobber list]* items each describe the effect of the instruction for the compiler. For example:

  ```
  asm( "movl %1, %%eax\n" "movl %%eax, %0":"=r" (x) : "r" (y) : "%eax" );
  ```

  where "=r" (x) is an output operand
  "r" (y) is an input operand.
  "%eax" is the clobber list consisting of one register, "%eax".

  The notation for the output and input operands is a constraint string surrounded by quotes, followed by an expression, and surrounded by parentheses. The constraint string describes how the input and output operands are used in the asm "string". For example, "r" tells the compiler that the operand is a register. The "=" tells the compiler that the operand is write only, which means that a value is stored in an output operand's expression at the end of the asm statement.

  Each operand is referenced in the asm "string" by a percent "%" and its number. The first operand is number 0, the second is number 1, the third is number 2, and so on. In the preceding example, "%0" references the output operand, and "%1" references the input operand. The asm "string" also contains "% %eax", which references machine register "%eax". Hard coded registers like "%eax" should be specified in the clobber list to prevent conflicts with other instructions in the compiler's assembly-language output.

[output operands], *[input operands]*, and *[clobber list]* items are described in more detail in the following sections.

Output Operands

The *[output operands]* are an optional list of output constraint and expression pairs that specify the result(s) of the asm statement. An output constraint is a string that specifies how a result is delivered to the expression. For example, "=r" (x) says the output operand is a write-only register that stores its value in the "C" variable x at the end of the asm statement. An example follows:

```
int x;
void example()
{
 asm( "movl $0, %0" : "=r" (x) );
}
```

The previous example assigns 0 to the "C" variable x. For the function in this example, the compiler produces the following assembly. If you want to produce an assembly listing, compile the example with the pgcc -S compiler option:

```
example:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 8
 movl $0, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

In the generated assembly shown, notice that the compiler generated two statements for the asm statement at line number 5. The compiler generated "*movl $0, %eax*" from the asm "*string*". Also notice that *%eax* appears in place of "*%0*" because the compiler assigned the *%eax* register to variable *x*. Since item 0 is an output operand, the result must be stored in its expression (*x*). The instruction *movl %eax, x(%rip)* assigns the output operand to variable *x*.

In addition to write-only output operands, there are *read/write* output operands designated with a "+" instead of a "=". For example, "+*r*" (*x*) tells the compiler to initialize the output operand with variable x at the beginning of the asm statement.

To illustrate this point, the following example increments variable x by 1:

```
int x=1;
void example2()
{
 asm( "addl $1, %0" : "+r" (x) );
}
```

To perform the increment, the output operand must be initialized with variable x. The *read/write* constraint modifier ("+") instructs the compiler to initialize the output operand with its expression. The compiler generates the following assembly code for the example2() function:

```
example2:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
 movl x(%rip), %eax
 addl $1, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

From the example2() code, two extraneous moves are generated in the assembly: one movl for initializing the output register and a second movl to write it to variable x. To eliminate these moves, use a memory constraint type instead of a register constraint type, as shown in the following example:

```
int x=1;
void example2()
{
 asm( "addl $1, %0" : "+m" (x) );
}
```

The compiler generates a memory reference in place of a memory constraint. This eliminates the two extraneous moves. Because the assembly uses a memory reference to variable x, it does not have to move x into a register prior to the asm statement; nor does it need to store the result after the asm statement. Additional constraint types are found in "Additional Constraints," on page 185.

```
example2:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
 addl $1, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

The examples thus far have used only one output operand. Because extended asm accepts a list of output operands, asm statements can have more than one result, as shown in the following example:

```
void example4()
{
int x=1; int y=2;
asm( "addl $1, %1\n" "addl %1, %0": "+r" (x), "+m" (y) );
}
```

This example increments variable *y* by *1* then adds it to variable *x*. Multiple output operands are separated with a comma. The first output operand is item 0 ("%0") and the second is item 1 ("%1") in the asm "*string*". The resulting values for *x* and *y* are *4* and *3* respectively.

Input Operands

The *[input operands]* are an optional list of input constraint and expression pairs that specify what "C" values are needed by the asm statement. The input constraints specify how the data is delivered to the asm statement. For example, "*r*" *(x)* says that the input operand is a register that has a copy of the value stored in "C" variable *x*. Another example is "*m*" *(x)* which says that the input item is the *memory* location associated with variable *x*. Other constraint types are discussed in "Additional Constraints," on page 185. An example follows:

```
void example5()
{
int x=1;
int y=2;
int z=3;
asm( "addl %2, %1\n" "addl %2, %0" : "+r" (x), "+m" (y) : "r" (z) );
}
```

The previous example adds variable z, item 2, to variable x and variable y. The resulting values for x and y are 4 and 5 respectively.

Another type of input constraint worth mentioning here is the *matching constraint*. A matching constraint is used to specify an operand that fills both an input as well as an output role. An example follows:

```
int x=1;
void example6()
{
asm( "addl $1, %1"
 : "=r" (x)
 : "0" (x) );
}
```

The previous example is equivalent to the *example2()* function shown in "Output Operands," on page 181. The constraint/expression pair, "*0*" *(x)*, tells the compiler to initialize output item *0* with variable *x* at the beginning of the *asm* statement. The resulting value for *x* is 2. Also note that "*%1*" in the asm "*string*" means the same thing as "*%0*" in this case. That is because there is only one operand with both an input and an output role.

Matching constraints are very similar to the *read/write* output operands mentioned in "Output Operands," on page 181. However, there is one key difference between *read/write* output operands and *matching constraints*. The *matching constraint* can have an *input expression* that differs from its *output expression*.

The following example uses different values for the input and output roles:

```
int x;
int y=2;
void example7()
{
asm( "addl $1, %1"
 : "=r" (x)
 : "0" (y) );
}
```

The compiler generates the following assembly for example7():

```
example7:
..Dcfb0:
 pushq %rbp
..Dcfi0:
```

```
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 8
 movl y(%rip), %eax
 addl $1, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

Variable *x* gets initialized with the value stored in *y*, which is *2*. After adding *1*, the resulting value for variable *x* is *3*.

Because *matching constraints* perform an input role for an output operand, it does not make sense for the output operand to have the read/write ("+") modifier. In fact, the compiler disallows *matching constraints* with read/write output operands. The output operand must have a write only ("=") modifier.

Clobber List

The *[clobber list]* is an optional list of strings that hold machine registers used in the asm "*string*". Essentially, these strings tell the compiler which registers may be clobbered by the asm statement. By placing registers in this list, the programmer does not have to explicitly save and restore them as required in traditional inline assembly (described in "Inline Assembly," on page 179). The compiler takes care of any required saving and restoring of the registers in this list.

Each machine register in the [clobber list] is a string separated by a comma. The leading '%' is optional in the register name. For example, "%eax" is equivalent to "eax". When specifying the register inside the asm "string", you must include two leading '%' characters in front of the name (for example., "%%eax"). Otherwise, the compiler will behave as if a bad input/output operand was specified and generate an error message. An example follows:

```
void example8()
{
int x;
int y=2;
asm( "movl %1, %%eax\n"
 "movl %1, %%edx\n"
 "addl %%edx, %%eax\n"
 "addl %%eax, %0"
 : "=r" (x)
 : "0" (y)
 : "eax", "edx" );
}
```

This code uses two hard-coded registers, eax and edx. It performs the equivalent of 3*y and assigns it to x, producing a result of 6.

In addition to machine registers, the clobber list may contain the following special flags:

"cc"
: The asm statement may alter the condition code register.

"memory"
: The asm statement may modify memory in an unpredictable fashion.

When the "memory" flag is present, the compiler does not keep memory values cached in registers across the asm statement and does not optimize stores or loads to that memory. For example:

```
asm("call MyFunc":::"memory");
```

This asm statement contains a "memory" flag because it contains a call. The callee may otherwise clobber registers in use by the caller without the "memory" flag.

The following function uses extended asm and the "cc" flag to compute a power of 2 that is less than or equal to the input parameter n.

```
#pragma noinline
int asmDivideConquer(int n)
{
 int ax = 0;
 int bx = 1;
 asm (
 "LogLoop:\n"
 "cmp %2, %1\n"
 "jnle Done\n"
 "inc %0\n"
 "add %1,%1\n"
 "jmp LogLoop\n"
 "Done:\n"
 "dec %0\n"
 :"+r" (ax), "+r" (bx) : "r" (n) : "cc");
 return ax;
}
```

The "cc" flag is used because the asm statement contains some control flow that may alter the condition code register. The #pragma noinline statement prevents the compiler from inlining the asmDivideConquer() function. If the compiler inlines asmDivideConquer(), then it may illegally duplicate the labels LogLoop and Done in the generated assembly.

Additional Constraints

Operand constraints can be divided into four main categories:

- Simple Constraints
- Machine Constraints
- Multiple Alternative Constraints
- Constraint Modifiers

Simple Constraints

The simplest kind of constraint is a string of letters or characters, known as *Simple Constraints*, such as the "r" and "m" constraints introduced in “Output Operands,” on page 181. Table 15.1, “Simple Constraints” describes these constraints.

Table 15.1. Simple Constraints

Constraint	Description
whitespace	Whitespace characters are ignored.
E	An immediate floating point operand.
F	Same as "E".
g	Any general purpose register, memory, or immediate integer operand is allowed.
i	An immediate integer operand.
m	A memory operand. Any address supported by the machine is allowed.
n	Same as "i".
o	Same as "m".
p	An operand that is a valid memory address. The expression associated with the constraint is expected to evaluate to an address (for example, "p" (&x)).
r	A general purpose register operand.
X	Same as "g".
0,1,2,..9	Matching Constraint. See “Input Operands,” on page 183 for a description.

The following example uses the general or "g" constraint, which allows the compiler to pick an appropriate constraint type for the operand; the compiler chooses from a general purpose register, memory, or immediate operand. This code lets the compiler choose the constraint type for "y".

```
void example9()
{
 int x, y=2;
 asm( "movl %1, %0\n" : "=r"
(x) : "g" (y) );
}
```

This technique can result in more efficient code. For example, when compiling example9() the compiler replaces the load and store of y with a constant 2. The compiler can then generate an immediate 2 for the y operand in the example. The assembly generated by pgcc for our example is as follows:

```
example9:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 3
 movl $2, %eax
## lineno: 6
 popq %rbp
 ret
```

In this example, notice the use of $2 for the "y" operand.

Of course, if y is always 2, then the immediate value may be used instead of the variable with the "i" constraint, as shown here:

```
void example10()
{
int x;
asm( "movl %1, %0\n"
 : "=r" (x)
 : "i" (2) );
}
```

Compiling example10() with pgcc produces assembly similar to that produced for example9().

Machine Constraints

Another category of constraints is *Machine Constraints*. The x86 and x86_64 architectures have several classes of registers. To choose a particular class of register, you can use the x86/x86_64 machine constraints described in Table 15.2, "x86/x86_64 Machine Constraints".

Table 15.2. x86/x86_64 Machine Constraints

Constraint	Description
a	a register (e.g., %al, %ax, %eax, %rax)
A	Specifies a or d registers. This is used primarily for holding 64-bit integer values on 32 bit targets. The d register holds the most significant bits and the a register holds the least significant bits.
b	b register (e.g, %bl, %bx, %ebx, %rbx)
c	c register (e.g., %cl, %cx, %ecx, %rcx)
C	Not supported.
d	d register (e.g., %dl, %dx, %edx, %rdx)
D	di register (e.g., %dil, %di, %edi, %rdi)
e	Constant in range of 0xffffffff to 0x7fffffff
f	Not supported.
G	Floating point constant in range of 0.0 to 1.0.
I	Constant in range of 0 to 31 (e.g., for 32-bit shifts).
J	Constant in range of 0 to 63 (e.g., for 64-bit shifts)
K	Constant in range of 0 to 127.
L	Constant in range of 0 to 65535.
M	Constant in range of 0 to 3 constant (e.g., shifts for lea instruction).
N	Constant in range of 0 to 255 (e.g., for out instruction).
q	Same as "r" simple constraint.
Q	Same as "r" simple constraint.
R	Same as "r" simple constraint.
S	si register (e.g., %sil, %si, %edi, %rsi)
t	Not supported.

Constraint	Description
u	Not supported.
x	XMM SSE register
y	Not supported.
Z	Constant in range of 0 to 0x7fffffff.

The following example uses the "x" or XMM register constraint to subtract c from b and store the result in a.

```
double example11()
{
 double a;
 double b = 400.99;
 double c = 300.98;
 asm ( "subpd %2, %0;"
 :"=x" (a)
 : "0" (b), "x" (c)
 );
 return a;
}
```

The generated assembly for this example is this:

```
example11:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 4
 movsd .C00128(%rip), %xmm1
 movsd .C00130(%rip), %xmm2
 movapd %xmm1, %xmm0
 subpd %xmm2, %xmm0;
## lineno: 10
## lineno: 11
 popq %rbp
 ret
```

If a specified register is not available, the pgcc and pgcpp compilers issue an error message. For example, pgcc and pgcpp reserves the "%ebx" register for Position Independent Code (PIC) on 32-bit system targets. If a program has an asm statement with a "b" register for one of the operands, the compiler will not be able to obtain that register when compiling for 32-bit with the -fPIC switch (which generates PIC).

To illustrate this point, the following example is compiled for a 32-bit target using PIC:

```
void example12()
{
int x=1;
int y=1;
asm( "addl %1, %0\n"
 : "+a" (x)
 : "b" (y) );
}
```

Compiling with the "-tp p7" switch chooses a 32-bit target.

```
% pgcc example12.c -fPIC -c -tp p7
PGC-S-0354-Can't find a register in class 'BREG' for extended ASM
operand 1 (example12.c: 3)
PGC/x86 Linux/x86 Rel Dev: compilation completed with severe errors
```

Multiple Alternative Constraints

Sometimes a single instruction can take a variety of operand types. For example, the x86 permits register-to-memory and memory-to-register operations. To allow this flexibility in inline assembly, use *multiple alternative constraints*. An *alternative* is a series of constraints for each operand.

To specify multiple alternatives, separate each alternative with a comma.

Table 15.3. Multiple Alternative Constraints

Constraint	Description
,	Separates each alternative for a particular operand.
?	Ignored
!	Ignored

The following example uses multiple alternatives for an add operation.

```
void example13()
{
int x=1;
int y=1;
asm( "addl %1, %0\n"
 : "+ab,cd" (x)
 : "db,cam" (y) );
}
```

example13() has two alternatives for each operand: "ab,cd" for the output operand and "db,cam" for the input operand. Each operand must have the same number of alternatives; however, each alternative can have any number of constraints (for example, the output operand in *example13()* has two constraints for its second alternative and the input operand has three for its second alternative).

The compiler first tries to satisfy the left-most alternative of the first operand (for example, the output operand in *example13()*). When satisfying the operand, the compiler starts with the left-most constraint. If the compiler cannot satisfy an alternative with this constraint (for example, if the desired register is not available), it tries to use any subsequent constraints. If the compiler runs out of constraints, it moves on to the next alternative. If the compiler runs out of alternatives, it issues an error similar to the one mentioned in *example12()*. If an alternative is found, the compiler uses the same alternative for subsequent operands. For example, if the compiler chooses the "c" register for the output operand in example13(), then it will use either the "a" or "m" constraint for the input operand.

Constraint Modifiers

Characters that affect the compiler's interpretation of a constraint are known as *Constraint Modifiers*. Two constraint modifiers, the "=" and the "+", were introduced in “Output Operands,” on page 181. Table 15.4 summarizes each constraint modifier.

Table 15.4. Constraint Modifier Characters

Constraint Modifier	Description
=	This operand is write-only. It is valid for output operands only. If specified, the "=" must appear as the first character of the constraint string.
+	This operand is both read and written by the instruction. It is valid for output operands only. The output operand is initialized with its expression before the first instruction in the asm statement. If specified, the "+" must appear as the first character of the constraint string.
&	A constraint or an alternative constraint, as defined in "Multiple Alternative Constraints," on page 189, containing an "&" indicates that the output operand is an *early clobber operand*. This type operand is an output operand that may be modified before the asm statement finishes using all of the input operands. The compiler will not place this operand in a register that may be used as an input operand or part of any memory address.
%	Ignored.
#	Characters following a "#" up to the first comma (if present) are to be ignored in the constraint.
*	The character that follows the "*" is to be ignored in the constraint.

The "=" and "+" modifiers apply to the operand, regardless of the number of alternatives in the constraint string. For example, the "+" in the output operand of example13() appears once and applies to both alternatives in the constraint string. The "&", "#", and "*" modifiers apply only to the alternative in which they appear.

Normally, the compiler assumes that input operands are used before assigning results to the output operands. This assumption lets the compiler reuse registers as needed inside the asm statement. However, if the asm statement does not follow this convention, the compiler may indiscriminately clobber a result register with an input operand. To prevent this behavior, apply the early clobber "&" modifier. An example follows:

```
void example15()
{
int w=1;
int z;
asm( "movl $1, %0\n"
 "addl %2, %0\n"
 "movl %2, %1"
 : "=a" (w), "=r" (z) : "r" (w) );
}
```

The previous code example presents an interesting ambiguity because "w" appears both as an output and as an input operand. So, the value of "z" can be either 1 or 2, depending on whether the compiler uses the same register for operand 0 and operand 2. The use of constraint "r" for operand 2 allows the compiler to pick any general purpose register, so it may (or may not) pick register "a" for operand 2. This ambiguity can be eliminated by changing the constraint for operand 2 from "r" to "a" so the value of "z" will be 2, or by adding an early clobber "&" modifier so that "z" will be 1. The following example shows the same function with an early clobber "&" modifier:

```
void example16()
{
int w=1;
int z;
asm( "movl $1, %0\n"
 "addl %2, %0\n"
 "movl %2, %1"
 : "=&a" (w), "=r" (z) : "r" (w) );
}
```

Adding the early clobber "&" forces the compiler not to use the "a" register for anything other than operand 0. Operand 2 will therefore get its own register with its own copy of "w". The result for "z" in *example16()* is 1.

Operand Aliases

Extended asm specifies operands in assembly strings with a percent '%' followed by the operand number. For example, "%0" references operand 0 or the output item "=&a" (w) in function *example16()* in the previous example. Extended asm also supports operand aliasing, which allows use of a symbolic name instead of a number for specifying operands, as illustrated in this example:

```
void example17()
{
int w=1, z=0;
asm( "movl $1, %[output1]\n"
 "addl %[input], %[output1]\n"
 "movl %[input], %[output2]"
 : [output1] "=&a" (w), [output2] "=r"
(z)
 : [input] "r" (w));
}
```

In *example17()*, "%[output1]" is an alias for "%0", "%[output2]" is an alias for "%1", and "%[input]" is an alias for "%2". Aliases and numeric references can be mixed, as shown in the following example:

```
void example18()
{
int w=1, z=0;
asm( "movl $1, %[output1]\n"
 "addl %[input], %0\n"
 "movl %[input], %[output2]"
 : [output1] "=&a" (w), [output2] "=r" (z)
 : [input] "r" (w));
}
```

In *example18()*, "%0" and "%[output1]" both represent the output operand.

Assembly String Modifiers

Special character sequences in the assembly string affect the way the assembly is generated by the compiler. For example, the "%" is an escape sequence for specifying an operand, "%%" produces a percent for hard coded registers, and "\n" specifies a new line. Table 15.5, "Assembly String Modifier Characters"summarizes these modifiers, known as *Assembly String Modifiers*.

Table 15.5. Assembly String Modifier Characters

Modifier	Description
\	Same as \ in printf format strings.
%*	Adds a '*' in the assembly string.
%%	Adds a '%' in the assembly string.
%A	Adds a '*' in front of an operand in the assembly string. (For example, %A0 adds a '*' in front of operand 0 in the assembly output.)
%B	Produces the byte op code suffix for this operand. (For example, %b0 produces 'b' on x86 and x86_64.)
%L	Produces the word op code suffix for this operand. (For example, %L0 produces 'l' on x86 and x86_64.)
%P	If producing Position Independent Code (PIC), the compiler adds the PIC suffix for this operand. (For example, %P0 produces @PLT on x86 and x86_64.)
%Q	Produces a quad word op code suffix for this operand if is supported by the target. Otherwise, it produces a word op code suffix. (For example, %Q0 produces 'q' on x86_64 and 'l' on x86.)
%S	Produces 's' suffix for this operand. (For example, %S0 produces 's' on x86 and x86_64.)
%T	Produces 't' suffix for this operand. (For example, %S0 produces 't' on x86 and x86_64.)
%W	Produces the half word op code suffix for this operand. (For example, %W0 produces 'w' on x86 and x86_64.)
%a	Adds open and close parentheses ( ) around the operand.
%b	Produces the byte register name for an operand. (For example, if operand 0 is in register 'a', then %b0 will produce '%al'.)
%c	Cuts the '$' character from an immediate operand.
%k	Produces the word register name for an operand. (For example, if operand 0 is in register 'a', then %k0 will produce '%eax'.)
%q	Produces the quad word register name for an operand if the target supports quad word. Otherwise, it produces a word register name. (For example, if operand 0 is in register 'a', then %q0 produces %rax on x86_64 or %eax on x86.)
%w	Produces the half word register name for an operand. (For example, if operand 0 is in register 'a', then %w0 will produce '%ax'.)
%z	Produces an op code suffix based on the size of an operand. (For example, 'b' for byte, 'w' for half word, 'l' for word, and 'q' for quad word.)
%+ %C %D %F %O %X %f %h %l %n %s %y are not supported.	

These modifiers begin with either a backslash "\" or a percent "%".

The modifiers that begin with a backslash "\" (e.g., "\n") have the same effect as they do in a printf format string. The modifiers that are preceded with a "%" are used to modify a particular operand.

These modifiers begin with either a backslash "\" or a percent "%" For example, "%b0" means, "produce the byte or 8 bit version of operand 0". If operand 0 is a register, it will produce a byte register such as %al, %bl, %cl, and so on.

Consider this example:

```
void example19()
{
int a = 1;
int *p = &a;
asm ("add%z0 %q1, %a0"
 : "=&p" (p) : "r" (a), "0" (p) );
}
```

On an x86 target, the compiler produces the following instruction for the asm string shown in the preceding example:

```
addl %ecx, (%eax)
```

The "%z0" modifier produced an 'l' (lower-case 'L') suffix because the size of pointer p is 32 bits on x86. The "%q1" modifier produced the word register name for variable a. The "%a0" instructs the compiler to add parentheses around operand 0, hence "(%eax)".

On an x86_64 target, the compiler produces the following instruction for the asm string shown in the preceding example:

```
addq %rcx, (%rax)
```

The "%z0" modifier produced a 'q' suffix because the size of pointer p is 64-bit on x86_64. Because x86_64 supports quad word registers, the "%q1" modifier produced the quad word register name (%rax) for variable a.

Extended Asm Macros

As with traditional inline assembly, described in"Inline Assembly," on page 179, extended asm can be used in a macro. For example, you can use the following macro to access the runtime stack pointer.

```
#define GET_SP(x) \
asm("mov %%sp, %0": "=m" (##x):: "%sp" );
void example20()
{
 void * stack_pointer;
 GET_SP(stack_pointer);
}
```

The GET_SP macro assigns the value of the stack pointer to whatever is inserted in its argument (for example, stack_pointer). Another "C" extension known as *statement expressions* is used to write the GET_SP macro another way:

```
#define GET_SP2 ({ \
void *my_stack_ptr; \
asm("mov %%sp, %0": "=m" (my_stack_ptr) :: "%sp" ); \
my_stack_ptr; \
})
```

```
void example21()
{
 void * stack_pointer = GET_SP2;
}
```

The statement expression allows a body of code to evaluate to a single value. This value is specified as the last instruction in the statement expression. In this case, the value is the result of the asm statement, my_stack_ptr. By writing an asm macro with a statement expression, the asm result may be assigned directly to another variable (for example, void * stack_pointer = GET_SP2) or included in a larger expression, such as: void * stack_pointer = GET_SP2 - sizeof(long).

Which style of macro to use depends on the application. If the asm statement needs to be a part of an expression, then a macro with a statement expression is a good approach. Otherwise, a traditional macro, like GET_SP(x), will probably suffice.

Intrinsics

Inline intrinsic functions map to actual x86 or x64 machine instructions. Intrinsics are inserted inline to avoid the overhead of a function call. The compiler has special knowledge of intrinsics, so with use of intrinsics, better code may be generated as compared to extended inline assembly code.

The PGI Workstation version 7.0 or higher compiler intrinsics library implements MMX, SSE, SS2, SSE3, SSSE3, SSE4a, ABM, and AVX instructions. The intrinsic functions are available to C and C++ programs on Linux and Windows. Unlike most functions which are in libraries, intrinsics are implemented internally by the compiler. A program can call the intrinsic functions from C/C++ source code after including the corresponding header file.

The intrinsics are divided into header files as follows:

Table 15.6. Intrinsic Header File Organization

Instructions	Header File		Instructions	Header File
ABM	intrin.h		SSE2	emmintrin.h
AVX	immintrin.h		SSE3	pmmintrin.h
MMX	mmintrin.h		SSSE3	tmmintrin.h
SSE	xmmintrin.h		SSE4a	ammintrin.h

The following is a simple example program that calls XMM intrinsics.

```
#include <xmmintrin.h>
int main(){
 __m128, __A, __B, result;
 __A = _mm_set_ps(23.3, 43.7, 234.234, 98.746);
 __B = _mm_set_ps(15.4, 34.3, 4.1, 8.6);
 result = _mm_add_ps(__A,__B);
 return 0;
}
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

NOTICE

TRADEMARKS

COPYRIGHT