PGI® COMPILERS & TOOLS

USER'S GUIDE FOR OPENPOWER CPUS

Version 2019

NVIDIA

TABLE OF CONTENTS

LIST OF TABLES

PREFACE

This guide is part of a set of manuals that describe how to use the PGI Fortran, C, and C++ compilers and program development tools. These compilers and tools include the *PGFORTRAN*, *PGC++*, *PGCC* compilers and the PGI profiler. They work in conjunction with an OpenPOWER assembler and linker. You can use the PGI compilers and tools to compile, debug, optimize, and profile serial and parallel applications for OpenPOWER 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. Neither guide teaches 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. You also 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 this PGI product. 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).

- *ISO/IEC 1539-1 : 2004, Information technology – Programming Languages – Fortran*, Geneva, 2004 (Fortran 2003).
- *ISO/IEC 1539-1 : 2010, Information technology – Programming Languages – Fortran*, Geneva, 2010 (Fortran 2008).
- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *The Fortran 2003 Handbook*, Adams et al, Springer, 2009.
- *OpenMP Application Program Interface*, Version 3.1, July 2011, 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).
- ISO/IEC 9899:2011, Information Technology – Programming Languages – C, Geneva, 2011 (C11).
- ISO/IEC 14882:2011, Information Technology – Programming Languages – C++, Geneva, 2011 (C++11).

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 guide contains the essential information on how to use the compiler and is divided into these sections:

Getting Started provides an introduction to the PGI compilers and describes their use and overall features.

Use Command-line Options provides an overview of the command-line options as well as task-related lists of options.

Optimizing and Parallelizing describes standard optimization techniques that, with little effort, allow users to significantly improve the performance of programs.

Using Function Inlining describes how to use function inlining and shows how to create an inline library.

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.

Using MPI describes how to use MPI with PGI products.

Using an Accelerator describes how to use the PGI Accelerator compilers.

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

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

Using Environment Variables describes the environment variables that affect the behavior of the PGI compilers.

Distributing Files – Deployment describes the deployment of your files once you have built, debugged and compiled them successfully.

Inter-language Calling provides examples showing how to place C language calls in a Fortran program and Fortran language calls in a C program.

Hardware and Software Constraints

This guide describes versions of the PGI compilers that produce assembly code for OpenPOWER 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

This guide uses the following conventions:

italic
: is used for emphasis.

`Constant Width`
: is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
: is used for commands.

[item1]
: in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ 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 a reduced fixed point size.

C/C++
C/C++ language statements are shown in the test of this guide using a reduced fixed point size.

The PGI compilers and tools are supported on a wide variety of Linux, macOS and Windows operating systems running on 64-bit x86-compatible processors, and on Linux running on OpenPOWER processors. (Currently, the PGI debugger is supported on x86-64/x64 only.) See the Compatibility and Installation section on the PGI website at https://www.pgroup.com/products/index.htm?tab=compat for a comprehensive listing of supported platforms.

Support for 32-bit development was deprecated in PGI 2016 and is no longer available as of the PGI 2017 release. PGI 2017 is only available for 64-bit operating systems and does not include the ability to compile 32-bit applications for execution on either 32- or 64-bit operating systems.

Terms

A number of terms related to systems, processors, compilers and tools are used throughout this guide. For example:

accelerator	FMA	-mcmodel=small	shared library
AVX	host	MPI	SIMD
CUDA	large arrays	MPICH	static linking
device	license keys	multicore	
driver	LLVM	NUMA	
DWARF	-mcmodel=medium	OpenPOWER	

For a complete definition of these terms and other terms in this guide with which you may be unfamiliar, please refer to the PGI online glossary at pgicompilers.com/definitions.

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
PGFORTRAN	ISO/ANSI Fortran 2003	pgfortran
PGCC	*ISO/ANSI* C11 and K&R C	pgcc
PGC++	*ISO/ANSI* C++17 with GNU compatibility	pgc++
PGI Profiler	Performance profiler	pgprof

In general, the designation *PGI Fortran* is used to refer to the PGI Fortran 2003 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 *PGC++* and *PGCC* is consistent with *PGFORTRAN*, though there are command-line options and features of these compilers that do not apply to *PGFORTRAN*, and vice versa.

Support for 32-bit development was deprecated in PGI 2016 and is no longer available as of the PGI 2017 release. PGI 2017 is only available for 64-bit operating systems and does not include the ability to compile 32-bit applications for execution on either 32-bit or 64-bit operating systems.

Related Publications

The following documents contain additional information related to the OpenPOWER architecture, and the compilers and tools available from The Portland Group.

- PGI Fortran Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19fortref-openpower.pdf describes the FORTRAN 77, Fortran 90/95, Fortran 2003 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.).
- *OpenPOWER ABI for Linux Supplement, Power Architecture 64-Bit ELF V2 ABI Specification*, http://openpowerfoundation.org/wp-content/uploads/2016/03/ABI64BitOpenPOWERv1.1_16July2015_pub4.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).

Chapter 1. GETTING STARTED

This section describes how to use the PGI compilers.

1.1. Overview

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 OpenPOWER processor-based system, regardless of whether the PGI compilers are installed on that system.

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. 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.

1.2. Creating an Example

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

hello

1. Create your program.

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

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

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:

   ```
   $ pgfortran hello.f
   ```

   By default, the executable output is placed in the file `a.out`. 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:

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

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:

   ```
   $ hello
   ```

   Below is the expected output:
   hello

1.3. Invoking the Command-level PGI Compilers

To translate and link a Fortran, C, or C++ program, the `pgfortran`, `pgcc` 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.

1.3.1. 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 Use 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.

1.3.2. 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 Use Command-Line Options.

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

- Command-line options and their arguments are case sensitive.
- 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.

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 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.

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

1.3.3. 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 Using OpenMP and Using Directives and Pragmas.

1.4. 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.

1.4.1. 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.

For systems with a case-insensitive file system, use the `-Mpreprocess` option, described in 'Command-Line Options Reference' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf, 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.F90`
indicates a Fortran 90/95 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
indicates an object file.
filename.a
indicates a library of object files.
filename.so
indicates a library of shared object files.

The driver passes files with `.s` extensions to the assembler and files with `.o`, `.so`, and `.a` 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 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 are 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 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 Create and Use Libraries.

1.4.2. Output Files

By default, an executable output file produced by one of the PGI compilers is placed in the file `a.out`. As the Hello example shows, you can use the -o option to specify the output file name.

If you use option -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 is 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 indicates the accepted input files.

Table 2 Option Descriptions

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 `pgc++`.	preprocessed file (`.f`)
-P	preprocessing	Source files. This option is not valid for `pgfortran`.	preprocessed file (`.i`)
-S	compilation	Source files or preprocessed files	assembly-language file (`.s`)
-c	assembly	Source files, or preprocessed files, or assembly-language files	unlinked object file (`.o`)
none	linking	Source files, or preprocessed files, assembly-language files, object files, or libraries	executable file (`a.out`)

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`
indicates a object file from the -c option.

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

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`, all of which are binary object files. Prior to compilation, the file `proto1.F` is preprocessed because it has a `.F` filename extension.

1.5. 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 OpenPOWER processor-based systems, refer to 'Fortran, C, and C++ Data Types' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

For more information on OpenPOWER-specific data representation, refer to the *OpenPOWER ABI for Linux Supplement, Power Architecture 64-Bit ELF V2 ABI Specification* listed in the 'Related Publications' section in the Preface.

1.6. Parallel Programming Using the PGI Compilers

The PGI compilers support many styles of parallel programming:

- Automatic shared-memory parallel programs compiled using the -Mconcur option to `pgfortran`, `pgcc` or `pgc++`. 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 `pgfortran`, `pgcc` or `pgc++`. 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. 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.

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 multicore 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 multicore processor is treated as a single CPU.

1.6.1. Run 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 OMP_NUM_THREADS environment variable to the desired number of processors. For information on how to set environment variables, refer to Setting Environment Variables.

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

1.7. Platform-specific considerations

The OpenPOWER Linux platform is supported by the PGI compilers and tools:

1.7.1. 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 PGI 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 `$PGI/linuxpower/include` and in `$PGI/linuxpower/include --gcc*`, such as `float.h`, `machine/_types.h`, `stddef.h`, `sys/cdefs.h` and others. Also, PGI's version of `stdarg.h` supports changes in newer versions of Linux.

If you are using the PGI 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.

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
```

1.8. 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.

1.8.1. Use 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.

1.8.2. 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, these files are named `.mypgf90rc`, `.mypgccrc`, and `.mypgc++rc`.

The following examples show how you can use these rc files to tailor a given installation for a particular purpose.

Table 3 Examples of Usine siterc and User rc Files

To do this...	Add the line shown to the indicated file(s)
Make available to all linuxpower compilations the libraries found in `/opt/newlibs/64`	**`set SITELIB=/opt/newlibs/64;`** to `/opt/pgi/linuxpower/19.1/bin/siterc`

To do this...	Add the line shown to the indicated file(s)
Add to all linuxpower compilations a new library path: /opt/local/fast	**append SITELIB=/opt/local/fast;** to /opt/pgi/linuxpower/19.1/bin/siterc
With linuxpower compilations, change -Mmpi to link in /opt/mympi/64/libmpix.a	**set MPILIBDIR=/opt/mympi/64;** **set MPILIBNAME=mpix;** to /opt/pgi/linuxpower/19.1/bin/siterc
With linuxpower compilations, always add -DIS64BIT -DIBM	**set SITEDEF=IS64BIT IBM;** to /opt/pgi/linuxpower/19.1/bin/siterc
Build an F90 or F95 executable for linuxpower or linuxpower that resolves PGI shared objects in the relative directory ./REDIST	**set set RPATH=./REDIST;** to ~/.mypgfortranrc **Note.**This only affects the behavior of PGFORTRAN for the given user.

1.9. 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 Use 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 Optimizing and 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 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 Using OpenMP and 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 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 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 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.

Chapter 2.
USE COMMAND-LINE OPTIONS

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

For a complete list of command-line options, their descriptions and use, refer to the 'Command-Line Options Reference' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

2.1. 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 Help with Command-line Options and Frequently-used Options.

2.1.1. 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.

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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf contains this information.

2.1.2. 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
```

2.1.3. 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.

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

The conflicting options 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 the following example.

Example: 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
```

2.2. 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:

  ```
  -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:

  ```
  -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]
  ```

- 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>
  ```

  For a complete description of subgroups, refer to the `-help` description in the *Command-line Options Reference* section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

2.3. 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.

2.3.1. Using -fast

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 option -fast. These options create a generally optimal set of flags. They incorporate optimization options to enable use of vector streaming SIMD

instructions. They enable vectorization with SIMD instructions, cache alignment, and flush to zero mode.

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

The following table shows the typical -fast options.

Table 4 Typical -fast Options

Use this option...	To do this...
-O2	Specifies a code optimization level of 2.
-Munroll=c:1	Unrolls loops, executing multiple instances of the original loop during each iteration.
-Mnoframe	Indicates to not generate code to set up a stack frame. **Note.** With this option, a stack trace does not work.
-Mlre	Indicates loop-carried redundancy elimination.
-Mpre	Indicates partial redundancy elimination

-fast typically includes the options shown in this table:

Table 5 Additional -fast Options

Use this option...	To do this...
-Mvect=simd	Generates packed SIMD instructions.
-Mcache_align	Aligns long objects on cache-line boundaries.
-Mflushz	Sets flush-to-zero mode.
-M[no]vect	Controls automatic vector pipelining.

For best performance on processors that support SIMD instructions, use the PGFORTRAN compiler and the -fast option.

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

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

2.3.2. Other Performance-Related Options

While -fast is designed to be the quickest route to best performance, it is 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. -Mipa=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, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf . These options include, but are not limited to, -Mvect, -Munroll, -Minline, -Mconcur, -Mpfi and -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 Optimizing and Parallelizing. For specific information about these options, refer to the 'Optimization Controls' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf .

2.4. 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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf. Also, there are a number of suboptions available with each of the `-M` options listed. For more information on those options, refer to the specific section on 'M Options by Category'.

Table 6 Commonly Used Command-Line Options

Use this option...	To do this...
`-fast`	This options creates a generally optimal set of flags for targets that support SIMD capability. It incorporates optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization with SIMD instructions, cache aligned and flushz.
`-g`	Instructs the compiler to include symbolic debugging information in the object module; sets the optimization level to zero unless a -O option is present on the command line.
`--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.

Use this option...	To do this...
-mcmodel=medium	Enables medium=model core generation for 64-bit targets, which is 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.
-Wl, <option>	Compiler driver passes the specified options to the linker.

Chapter 3. OPTIMIZING AND 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.

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 these optimization options:

-fast	-Minline	-O	-Munroll
-Mconcur	-Mvect	-Minfo	-Mneginfo
-Msafeptr			

-fast	-Minline	-O	-Munroll
-Mconcur	-Mpfi	-Mvect	-Minfo
-Mneginfo	-Mpfo		

This chapter also describes how to choose optimization options to use with the PGI compilers. This overview is helpful 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* section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf .

3.1. 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 OpenPOWER architecture, instruction set and registers.

For discussion purposes, we categorize optimization:

3.1.1. Local Optimization

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. Local optimization is performed on a block-by-block basis within a program’s basic blocks.

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.

3.1.2. Global Optimization

This optimization is performed on a subprogram/function 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 ad hoc branches such as IFs or 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.

3.1.3. 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 vector 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.

3.1.4. 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.

3.1.5. 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.

3.2. Getting Started with Optimization

The first concern should be getting the program to execute and produce correct results. To get the program running, start by compiling and linking without optimization. Add -O0 to the compile line to select no optimization; or add -g to debug the program easily and isolate any coding errors exposed during porting to OpenPOWER platform.

To get started quickly with optimization, a good set of options to use with any of the PGI compilers is -fast . For example:

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

For all of the PGI Fortran, C, and C++ compilers, the -fast -Mipa=fast,inline 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,inline option invokes interprocedural analysis (IPA), including several IPA suboptions. The inline suboption enables automatic inlining with IPA. If you do not wish to use automatic inlining, you can compile with -Mipa=fast and use several IPA suboptions without inlining.

Aggregate options incorporate a generally optimal set of flags that enable use of SIMD instructions .

The following table shows the typical -fast options.

Table 7 Typical -fast Options

Use this option...	To do this...
-O2	Specifies a code optimization level of 2 and -Mvect=SIMD.
-Munroll=c:1	Unrolls loops, executing multiple instances of the original loop during each iteration.
-Mlre	Indicates loop-carried redundancy elimination.
-Mautoinline	Enables automatic function inlining in C & C++.

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.

There are other useful command line options related to optimization and parallelization, such as -help, -Minfo, -Mneginfo, -dryrun, and -v.

3.2.1. -help

As described in Help with Command-Line Options, 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:

```
-O Set opt level. All -O1 optimizations plus traditional scheduling and
  global scalar optimizations performed
```

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:

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

3.2.2. -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 standard error (stderr) as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, vector 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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

3.2.3. -Mneginfo

You can use the `-Mneginfo` option to display informational messages to standard error (stderr) that explain why certain optimizations are inhibited.

For more information on `-Mneginfo`, refer to 'Optimization Controls' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

3.2.4. -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 are printed to standard error (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.

3.2.5. -v

The `-v` option is similar to -dryrun, except each compilation step is performed and not simply printed.

3.2.6. PGI Profiler

The PGI profiler is a profiling tool that provides a way to visualize the performance of the components of your program. Using tables and graphs, the profiler associates execution time and resource utilization data with the source code and instructions of your program. This association allows you to see where a program's execution time is spent. Through resource utilization data and compiler analysis information, the profiler helps you to understand why certain parts of your program have high execution times. This information may help you with selecting which optimization options to use with your program.

The profiler also allows you to correlate the messages produced by `-Minfo` and `-Mneginfo`, described above, to your program's source code. This feature is known as the Common Compiler Feedback Format (CCFF).

For more information on the profiler, refer to the Profiler User's Guide, www.pgroup.com/resources/docs/19.1/pdf/pgi19profug.pdf.

3.3. 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 the PGI profiler to aid with your optimization, it 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.

3.4. Local and Global Optimization

This section describes local and global optimization.

3.4.1. -Msafeptr

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. There are a number of suboptions for `-Msafeptr`:

- `-Msafeptr=all` – All pointers are safe. Equivalent to the default setting: `-Msafeptr`.
- `-Msafeptr=arg` – Function formal argument pointers are safe. Equivalent to `-Msafeptr=dummy`.
- `-Msafeptr=global` – Global pointers are safe.
- `-Msafeptr=local` – Local pointers are safe. Equivalent to `-Msafeptr=auto`.
- `-Msafeptr=static` – Static local pointers are safe.

If your C/C++ program has pointer aliasing and you also want automating inlining, then compiling with `-Mipa=fast` or `-Mipa=fast,inline` includes pointer aliasing optimizations. IPA may be able to optimize some of the alias references in your program and leave intact those that cannot be safely optimizied.

3.4.2. -O

Using the PGI compiler commands with the `-O`<level> option (the capital O is for Optimize), you can specify any integer level from 0 to 4.

-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. To enable debugging, include `-g` on your compile line.

-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.

Types of Optimizations

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 specifies 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
Induction variable elimination
Invariant code motion

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.

The `-fast` option includes `-O2` on all 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
```

3.5. Loop Unrolling using -Munroll

This optimization unrolls loops, which 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 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.

Examples Showing Effect of Unrolling

The following side-by-side examples show the effect of code unrolling on a segment that computes a dot product.

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.

Table 8 Example of Effect of Code Unrolling

Dot Product Code	Unrolled 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`
	`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, you can precisely control whether and how a given loop is unrolled. For more information on `-Munroll`, refer to Use Command-line Options.

3.6. Vectorization using -Mvect

The -Mvect option is included as part of -fast on all 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 vector instructions on OpenPOWER processors where these are supported.

The -Mvect option can speed up code which contains well-behaved countable loops which operate on large floating point 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.

3.6.1. 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 Using Directives and Pragmas.

The vectorizer performs the following operations:

- Loop interchange
- Loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SIMD 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. The following table lists and briefly describes some of the -Mvect suboptions.

Table 9 -Mvect Suboptions

Use this option ...	To instruct the vectorizer to do this ...
-Mvect=altcode	Generate appropriate code for vectorized loops.
-Mvect=[no]assoc	Perform[disable] associativity conversions that can change the results of a computation due to a round-off error. 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.
-Mvect=cachesize:n	Tiles 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.
-Mvect=fuse	Enable loop fusion.
-Mvect=gather	Enable vectorization of indirect array references.
-Mvect=idiom	Enable idiom recognition.
-Mvect=levels:<n>	Set the maximum next level of loops to optimize.
-Mvect=nocond	Disable vectorization of loops with conditions.
-Mvect=partial	Enable partial loop vectorization via inner loop distribution.
-Mvect=prefetch	Automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SIMD instructions are not generated.
-Mvect=short	Enable short vector operations.
-Mvect=simd	Automatically generate packed SIMD, 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.
-Mvect=sizelimit:n	Limit the size of vectorized loops.
-Mvect=sse	Equivalent to -Mvect=simd.
-Mvect=tile	Enable loop tiling.
-Mvect=uniform	Perform consistent optimizations in both vectorized and residual loops. Be aware that this may affect the performance of the residual loop.

Inserting no in front of the option disables the option. For example, to disable the generation of vector instructions on OpenPOWER, compile with -Mvect=nosimd.

3.6.2. Vectorization Example Using SIMD Instructions

One of the most important vectorization options is -Mvect=simd. When you use this option, the compiler automatically generates SIMD vector 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.

In the program in Vector operation using SIMD instructions, the vectorizer recognizes the vector operation in subroutine 'loop' when either compiler switch -Mvect=simd or -fast is used. This example shows the compilation, informational messages, and runtime results using SIMD instructions on an IBM POWER9 system, along with issues that affect SIMD performance.

Loops vectorized using SIMD 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.

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 Vector operation using SIMD instructions both with and without the option -Mvect=simd .

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=nosimd disables SIMD vectorization:

```
% pgfortran -fast -Mvect=nosimd -Minfo vadd.f -Mfree -o vadd
vector_op:
      4, Loop unrolled 16 times
         Generated 1 prefetches in scalar loop
      9, Loop not vectorized/parallelized: contains call
loop:
     18, Loop unrolled 8 times
         FMA (fused multiply-add) instruction(s) generated
```

The following output shows a sample result if the generated executable is run and timed on an IBM POWER9 system:

```
$ /bin/time vadd
   -1.000000       -771.0000       -3618.000       -6498.000
   -9999.000
2.31user 0.00system 0:02.57elapsed 89%CPU (0avgtext+0avgdata 6976maxresident)k
8192inputs+0outputs (4major+149minor)pagefaults 0swaps
```

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

```
% pgfortran -fast -Minfo vadd.f -Mfree -o vadd
vector_op:
      4, Loop not vectorized: may not be beneficial
         Unrolled inner loop 8 times
         Residual loop unrolled 7 times (completely unrolled)
         Generated 1 prefetches in scalar loop
      9, Loop not vectorized/parallelized: contains call
loop:
     18, Generated 2 alternate versions of the loop
         Generated vector simd code for the loop
         Generated 3 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 3 prefetch instructions for the loop
         Generated vector simd code for the loop
         Generated 3 prefetch instructions for the loop
         FMA (fused multiply-add) instruction(s) generated
```

Notice the informational messages for the loop at line 18. The first line of the message indicates that two alternate versions of the loop were generated. The loop count and alignments of the arrays determine which of these versions is executed. The next several lines indicate the loop was vectorized and 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-simd
   -1.000000       -771.0000       -3618.000       -6498.000
   -9999.000
0.62user 0.00system 0:00.65elapsed 95%CPU (0avgtext+0avgdata 6976maxresident)k
0inputs+0outputs (0major+151minor)pagefaults 0swaps
```

The SIMD result is 3.7 times faster than the equivalent non-SIMD 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 SIMD instructions is most effective.

- If data is aligned properly, performance will be better in general than when using SIMD operations on unaligned data.
- If the compiler can guarantee that data is aligned properly, even more efficient sequences of SIMD instructions can be generated.
- The efficiency of loops that operate on single-precision data can be higher. SIMD instructions can operate on four single-precision elements concurrently, but only two double-precision elements.

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).

3.7. 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. For a complete specification of `-Mconcur`, refer to the 'Optimization Controls' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

A loop is considered parallelizable if it 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 not to parallelize innermost loops, since these often by definition are vectorizable and it is seldom profitable to both vectorize and parallelize the same loop, especially on multicore processors. Compiler switches and directives are available to let you override most of these restrictions on auto-parallelization.

3.7.1. 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 parallelizer. In addition, these parallelizer operations can be controlled from within code using directives and pragmas. For details on the use of directives and pragmas, refer to 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. The following table lists and briefly describes some of the `-Mconcur` suboptions.

Table 10 -Mconcur Suboptions

Use this option ...	To instruct the parallelizer to do this...
`-Mconcur=allcores`	Use all available cores. Specify this option at link time.
`-Mconcur=[no]altcode`	Generate [do not 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. Specifying `noaltcode` disables this option and no alternate serial code is generated.
`-Mconcur=[no]assoc`	Enable [disable] parallelization of loops with associative reductions.
`-Mconcur=bind`	Bind threads to cores. Specify this option at link time.
`-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 occurs, and last values of scalars are assumed to be safe.
`-Mconcur=dist:{block|cyclic}`	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.
`-Mconcur=innermost`	Enable parallelization of innermost loops.
`-Mconcur=levels:<n>`	Parallelize loops nested at most n levels deep.
`-Mconcur=[no]numa`	Use thread/processors affinity when running on a NUMA architecture. Specifying `-Mconcur=nonuma` disables this option.

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 are 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 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 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.

3.7.2. 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 section, 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
1    a(i) = b(i) + c(i)
 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 does 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 loops in C/C++ and Fortran:

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

f(v);
```

```
! Fortran version
do I = 1,N
  if (x(I) > 5.0 ) then
    t = x(I)
  endif
enddo
v = t

call f(v)
```

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 C/C++ and Fortran examples.

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

```
! Fortran version
do I = 1,N
 if (x(I)>0.0) then
  t=2.0
 else
  t=3.0
  y(i)=t
 endif
enddo
v=t
```

Notice that `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 loops in which each use of `t` within the loop is reached by a definition from the same iteration.

```
/* C/C++ Version */
for (i=1;i<N;i++){
 if(x[i]>0.0){
```

```
  t=x[i];
  y[i]=t;
 }
}
v=t;

f(v);
```

```
! Fortran Version
do I = 1,N
 if (x(I)>0.0) then
  t=x(I)
  y(i)=t
 endif
enddo
v=t

call f(v)
```

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 directive or pragma `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.

```
!pgi$l safe_lastval    ! Fortran Version
```

```
#pragma loop safe_lastval     /* C/C++ Version */
```

The resulting code looks similar to this:

```
/* C/C++ Version */
#pragma loop safe_lastval
...
for (i=1;i<N;i++){
 if(x[i]>5.0 ) t=x[i];
}
 v = t;
```

```
! Fortran Version
!pgi$l safe_lastv
...
do I = 1,N
 if (x(I) > 5.0 ) then
  t = x(I)
 endif
enddo
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.

3.8. 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 11 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
-O<level>	none or -g	none	level
-O<level> <= 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. The `-fast` option sets the optimization level to a target-dependent optimization level if no `-O` options are supplied.

3.9. 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.

Using Directives and Pragmas provides details on how to add directives and pragmas to your source files.

3.10. 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 PGFORTRAN.
- The SYSTEM_CLOCK or CPU_CLOCK intrinsics in PGF95 or PGFORTRAN.

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 a Fortran program unit.

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
```

3.11. 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 multicore 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 64-bit Linux systems.

When a program is linked with the system `libnuma` library, the program depends on that library to run. On systems without a `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`.

3.11.1. 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 function inlining** – In C/C++, you can inline static functions with the `inline` keyword by using the `-Mautoinline` option, which is included with `-fast`.
- **Function inlining** – You can inline functions which were extracted to the inline libraries in C/Fortran/C++. There are two ways of enabling function inlining: with and without the `lib` suboption. For the latter, you create inline libraries, for example using the `pgfortran` compiler driver and the `-o` and `-Mextract` options.

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

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

```
-Mautoinline
-Mextract
-Minline
-Mnoinline
-Mrecursive
```

4.1. Automatic function inlining in C/C++

To enable automatic function inlining in C/C++ for static functions with the `inline` keyword, use the `-Mautoinline` option (included in `-fast`). Use `-Mnoautoinline` to disable it.

Several `-Mautoinline` suboptions let you determine the selection criteria. These suboptions are:

maxsize:n
Automatically inline functions size `n` and less

totalsize:n
Limit automatic inlining to total size of `n`

4.2. 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.

[maxsize:]`number`
A numeric option is assumed to be a size. Functions of size `number` or less are inlined. If both `number` and `function` are specified, then functions matching the given name(s) or meeting the size requirements are inlined.

reshape
Fortran subprograms with array arguments are not inlined by default if the array shape does not match the shape in the caller. Use this option to override the default.

smallsize:`number`
Always inline functions of size smaller than `number` regardless of other size limits.

totalsize:`number`
Stop inlining in a function when the function's total inlined size reaches the `number` specified.

[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 maxsize 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.

Inlining can be disabled with `-Mnoinline`.

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=maxsize:100 myprog.f
```

For more information on the `-Minline` options, refer to '-M Options by Category' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

4.3. 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
```

4.4. 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.

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.

4.4.1. 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.

4.4.2. 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.

4.4.3. 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.

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 $(SRC)/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
```

4.5. 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
```

4.6. 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).

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=maxsize:10
```

4.7. 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
- Functions 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 PGFORTRAN Fortran compiler supports the OpenMP Fortran Application Program Interface. The PGCC and PGC++ 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 section provides information on OpenMP as it is supported by PGI compilers. Currently, all PGI compilers support the version 3.1 OpenMP specification.

Use the -mp compiler switch to enable processing of the OpenMP pragmas listed in this section. As of the PGI 2011 Release, the OpenMP runtime library is linked by default. Note that GNU pthreads are not completely interoperable with OpenMP threads.

When using pgc++ on Linux, the GNU STL is thread-safe to the extent listed in the GNU documentation as required by the C++11 standard. If an STL thread-safe issue is suspected, the suspect code can be run sequentially inside of an OpenMP region using `#pragma omp critical` sections.

This section describes how to use the following option supporting OpenMP: -mp

5.1. OpenMP Overview

OpenMP 3.1

The PGI Fortran, C, and C++ compilers support OpenMP 3.1 on all platforms.

OpenMP 4.5

The PGI Fortran, C, and C++ compilers compile most OpenMP 4.5 programs for parallel execution across all the cores of a multicore CPU or server. `target` regions are

implemented with default support for the multicore host as the target, and **parallel** and **distribute** loops are parallelized across all OpenMP threads.

Current limitations include:

- The **simd** construct can be used to provide tuning hints; the **simd** construct's **private**, **lastprivate**, **reduction**, and **collapse** clauses are processed and supported.
- The **declare simd** construct is ignored.
- The **ordered** construct's **simd** clause is ignored.
- The **task** construct's **depend** and **priority** clauses are not supported.
- The loop construct's **linear**, **schedule**, and **ordered(n)** clauses are not supported.
- The **declare reduction** directive is not supported.

5.1.1. 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.

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

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.

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, see the OpenMP website, http://www.openmp.org.

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

5.1.2. 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 both lexically and 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.

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.

5.1.3. OpenMP Example

Look at the following simple OpenMP example involving loops.

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
```

5.2. 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 or pragma plus a structured block.

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

5.3. 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 *$OMP.
- Must start in column 1 (one) for free-form code.
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is C$.

The *directive_name* can be any of the directives listed in Directive and Pragma Summary Table. The valid clauses depend on the directive. Directive and Pragma Clauses provides a list of clauses, the directives and pragmas 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.

5.4. C/C++ Parallelization Pragmas

Parallelization pragmas are #pragma statements in a C or C++ program that are interpreted by the PGI 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 rules:

- The pragmas follow the conventions of the C and C++ rules.
- 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.

5.5. Directive and Pragma Recognition

The compiler option `-mp` enables recognition of the parallelization directives and pragmas.

The use of this option also implies:

`-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.

5.6. Directive and Pragma Summary Table

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

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

5.6.1. Directive and Pragma Summary Table

Table 12 Directive and Pragma Summary Table

Fortran Directive and C++ Pragma	Description
ATOMIC [TYPE] ... END ATOMIC and atomic	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive or critical pragma. TYPE may be empty or one of the following: UPDATE, READ, WRITE, or CAPTURE. The END ATOMIC directive is only allowed when ending ATOMIC CAPTURE regions.  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.
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.

Fortran Directive and C++ Pragma	Description
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.

5.7. 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.

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

Table 13 Directive and Pragma 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

This clause...	Applies to this directive	Applies to this pragma	Has this functionality
			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.
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	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 loop construct or last section of an OpenMP `section`.

This clause...	Applies to this directive	Applies to this pragma	Has this functionality
	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	Eliminates the barrier implicit at the end of a parallel region.
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	Specifies that this block within the parallel DO or FOR region needs to be execute serially in the same order indicated by the enclosing loop.
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.

This clause...	Applies to this directive	Applies to this pragma	Has this functionality

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 in `list` 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 looping 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.

5.8. 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` header 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);
```

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 256 threads. The OpenPOWER compiler relies on the llvm OpenMP runtime, which has a maximum of 2^{31} threads.

The following table summarizes the runtime library calls.

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

Table 14 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);`
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)`

Runtime Library Routines with Examples	
	`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.	
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);`
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++	`int 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	

Runtime Library Routines with Examples	
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 stack_size);`
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++	`int omp_get_team_size(int level);`
omp_in_final	
Returns whether or not the call is within a final task. Returns `.TRUE.` for directives and non-zero for pragmas if called from within a final task region.	
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.	
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.	
Fortran	`logical function omp_get_dynamic()`
C/C++	`void omp_get_dynamic(void);`
omp_set_nested	
Allows enabling/disabling of nested parallel regions.	
Fortran	`subroutine omp_set_nested(nested)`

Runtime Library Routines with Examples	
	`logical nested`
C/C++	`void omp_set_nested(int nested);`
omp_get_nested	
Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled.	
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++	`void omp_set_schedule(omp_sched_t kind, int chunk_size);`
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++	`void omp_get_schedule(omp_sched_t *kind, int *chunk_size);`
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();`
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'`

Runtime Library Routines with Examples	
	`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++	`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`

C/C++	`int omp_test_lock(omp_lock_t *lock);` `int omp_test_nest_lock(omp_nest_lock_t *lock);`

5.9. 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 OpenPOWER compiler relies on the llvm OpenMP runtime, which has different default values.

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

Table 15 OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_MAX_ACTIVE_LEVELS	2^{31}	Specifies the maximum number of nested parallel regions.
OMP_NESTED	FALSE	Enables (TRUE) or disables (FALSE) nested parallelism.
OMP_NUM_THREADS	# of logical CPUs	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 0	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	2^{31}	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 on Linux using Open MPI. Of course, you may always build using an arbitrary version of MPI; to do this, use the `-I`, `-L`, and `-l` option.

PGI products for Linux include Open MPI, PGI products for macOS includes MPICH, and PGI products for Windows includes MS-MPI. This section 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.

6.1. MPI Overview

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

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.

6.2. Using Open MPI on Linux

PGI products for Linux ship with a PGI-built version of Open MPI that includes everything required to compile, execute and debug MPI programs using Open MPI.

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.

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

6.3. Using MPI Compiler Wrappers

When you use MPI compiler wrappers to build with the `-fpic` or `-mcmodel=medium` options, then you must specify `-pgf90libs` to link with the correct libraries. Here are a few examples:

For a static link to the MPI libraries, use this command:

```
% mpifort hello.f
```

For a dynamic link to the MPI libraries, use this command:

```
% mpifort hello.f -pgf90libs
```

To compile with `-fpic`, which, by default, invokes dynamic linking, use this command:

```
% mpifort -fpic -pgf90libs hello.f
```

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

```
% mpifort -mcmodel=medium -pgf90libs hello.f
```

6.4. 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 linuxpower 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.

6.5. Testing and Benchmarking

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

```
% cp -r $PGI/linuxpower/19.1/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 four 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 section 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*.

7.1. Overview

The programming model and directives described in this section 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, C, 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.

7.1.1. 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 section concentrates on specification of loops and regions of code to be offloaded to an accelerator.

7.1.2. Features Not Covered or Implemented

This section 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, this feature is not currently supported.

7.2. 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 section 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 structured block defined by an OpenACC compute construct. A *compute construct* 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. The dynamic range of a compute construct, including any code in procedures called from within the construct, is the compute region. In this release, compute regions may not contain other compute regions or data regions.

Construct
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 construct, such as device memory allocation or data movement between the host and device memory. Loops in a compute construct are targeted for execution on the accelerator. The dynamic range of a construct including any code in procedures called from within the construct, is called a *region*.

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

Data region
a region defined by an OpenACC data construct, or an implicit data region for a function or subroutine containing OpenACC 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.

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
the dynamic range of a construct, including any procedures invoked from within the construct.

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.

7.3. Execution Model

The execution model targeted by the PGI 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.

7.3.1. 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

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

7.3.2. 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.

7.4. 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.

7.4.1. 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.

7.4.2. 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.

7.4.3. 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, it is up to the programmer to manage these caches. However, in the OpenACC programming model, the compiler manages these caches using hints from the programmer in the form of directives.

7.4.4. CUDA Unified Memory

PGI compilers have supported the use of CUDA Unified Memory since PGI 17.7. This feature, described in detail in the OpenACC and CUDA Unified Memory, https://www.pgroup.com/blogs/posts/openacc-unified-memory.htm *PGInsider* blog, is available with the Linux/x86-64 and Linux/OpenPOWER compilers. It is supported on Linux/x86-64 using both the default PGI code generator and the LLVM-based code generator. To enable this feature, add the option `-ta=tesla:managed` to the compiler and linker command lines.

In the presence of `-ta=tesla:managed`, all C/C++/Fortran explicit allocation statements in a program unit are replaced by equivalent "managed" data allocation calls that place the data in CUDA Unified Memory. Managed data share a single address for CPU/GPU and data movement between CPU and GPU memories is implicitly handled by the CUDA driver. Therefore, OpenACC data clauses and directives are not needed for "managed" data. They are essentially ignored, and in fact can be omitted.

When a program allocates managed memory, it allocates host pinned memory as well as device memory thus making allocate and free operations somewhat more expensive and data transfers somewhat faster. A memory pool allocator is used to mitigate the overhead of the allocate and free operations. The pool allocator is enabled by default for `-ta=tesla:managed` or `-ta=tesla:pinned`.

Data movement of managed data is controlled by the NVIDIA CUDA GPU driver; whenever data is accessed on the CPU or the GPU, it could trigger a data transfer if the last time it was accessed was not on the same device. In some cases, page thrashing may occur and impact performance. An introduction to CUDA Unified Memory is available on Parallel Forall.

This feature has the following limitations:

- Use of managed memory applies only to dynamically-allocated data. Static data (C static and extern variables, Fortran module, common block and save variables) and function local data is still handled by the OpenACC runtime. Dynamically allocated Fortran local variables and Fortran allocatable arrays are implicitly managed but Fortran array pointers are not.
- Given an allocatable aggregate with a member that points to local, global or static data, compiling with `-ta=tesla:managed` and attempting to access memory through that pointer from the compute kernel will cause a failure at runtime.
- C++ virtual functions are not supported.
- The `-ta=tesla:managed` compiler option must be used to compile the files in which variables are allocated, even if there is no OpenACC code in the file.

This feature has the following additional limitations when used with NVIDIA Kepler GPUs:

- Data motion on Kepler GPUs is achieved through fast pinned asynchronous data transfers; from the program's perspective, however, the transfers are synchronous.
- The PGI runtime enforces synchronous execution of kernels when `-ta=tesla:managed` is used on a system with a Kepler GPU. This situation may result in slower performance because of the extra synchronizations and decreased overlap between CPU and GPU.
- The total amount of managed memory is limited to the amount of available device memory on Kepler GPUs.

CUDA Unified Memory Pool Allocator

Dynamic memory allocations are made using cudaMallocManaged(), a routine which has higher overhead than allocating non-unified memory using cudaMalloc(). The more calls to cudaMallocManaged(), the more significant the impact on performance.

To mitigate the overhead of cudaMallocManaged() calls, both `-ta=tesla:managed` and `-ta=tesla:pinned` use a CUDA Unified Memory pool allocator to minimize the number of calls to cudaMallocManaged(). The pool allocator is enabled by default. It can be disabled, or its behavior modified, using these environment variables:

Table 16 Pool Allocator Environment Variables

Environment Variable	Use
PGI_ACC_POOL_ALLOC	Disable the pool allocator. The pool allocator is enabled by default; to disable it, set PGI_ACC_POOL_ALLOC to 0.
PGI_ACC_POOL_SIZE	Set the size of the pool. The default size is 1GB but other sizes (i.e., 2GB, 100MB, 500KB, etc.) can be used. The actual pool size is set such that the size is the nearest, smaller number in the Fibonacci series compared to the provided or default size. If necessary, the pool allocator will add more pools but only up to the PGI_ACC_POOL_THRESHOLD value.
PGI_ACC_POOL_ALLOC_MAXSIZE	Set the maximum size for allocations. The default maximum size for allocations is 500MB but another size (i.e., 100KB, 10MB, 250MB, etc.) can be used as long as it is greater than or equal to 16B.
PGI_ACC_POOL_ALLOC_MINSIZE	Set the minimum size for allocation blocks. The default size is 128B but other sizes can be used. The size must be greater than or equal to 16B.
PGI_ACC_POOL_THRESHOLD	Set the percentage of total device memory that the pool allocator can occupy. Values from 0 to 100 are accepted. The default value is 50, corresponding to 50% of device memory.

7.5. OpenACC Programming Model

With the emergence of GPU and many-core architectures in high performance computing, programmers want the ability to program using a familiar, high level programming model that provides both high performance and portability to a wide range of computing architectures. OpenACC emerged in 2011 as a programming model that uses high-level compiler directives to expose parallelism in the code and parallelizing compilers to build the code for a variety of parallel accelerators.

This chapter will not attempt to describe OpenACC itself. For that, please refer to the OpenACC specification on the OpenACC www.openacc.org website. Here, we will discuss differences between the OpenACC specification and its implementation by the PGI compilers.

Other resources to help you with your parallel programming including video tutorials, course materials, code samples, a best practices guide and more are available on the OpenACC website.

7.5.1. Enable Accelerator Directives

PGI compilers enable accelerator directives with the `-acc` and `-ta` command line options. For more information on this option as it relates to the Accelerator, refer to Compiling an Accelerator Program.

_OPENACC macro

The `_OPENACC` 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 OpenACC directives supported by the implementation. For example, the version for November, 2017 is 201711. All OpenACC compilers define this macro when OpenACC directives are enabled.

7.5.2. Support

The PGI compilers implement OpenACC 2.6 as defined in *The OpenACC Application Programming Interface*, Version 2.6, November 2017, http://www.openacc.org, with the exception that the following features are not yet supported:

- nested parallelism
- declare link
- enforcement of the **`cache`** clause restriction that all references to listed variables must lie within the region being cached

7.5.3. Extensions

The PGI Fortran compiler supports an extension to the `collapse` clause on the `loop` construct. The OpenACC specification defines `collapse`:

```
collapse(n)
```

For Fortran, PGI supports the use of the identifier `force` within `collapse`:

```
collapse(force:n)
```

Using `collapse(force:n)` instructs the compiler to enforce collapsing parallel loops that are not perfectly nested.

7.6. Supported Processors and GPUs

This PGI release supports OpenPOWER host processors.

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

For more information on these flags as they relate to accelerator technology, refer to Compiling an Accelerator Program.

For a complete list of supported CUDA GPUs, refer to the NVIDIA website at: http://www.nvidia.com/object/cuda_learn_products.html

7.7. CUDA Toolkit Versions

The PGI compilers use NVIDIA's CUDA Toolkit when building programs for execution on an NVIDIA GPU. Every PGI installation package puts the required CUDA Toolkit components into a PGI installation directory called `2019/cuda`.

An NVIDIA CUDA driver must be installed on a system with a GPU before you can run a program compiled for the GPU on that system. PGI products do not contain CUDA Drivers. You must download and install the appropriate CUDA Driver from NVIDIA. The CUDA Driver version must be at least as new as the version of the CUDA Toolkit with which you compiled your code.

The PGI tool `pgaccelinfo` prints the driver version as its first line of output. You can use it to find out which version of the CUDA Driver is installed on your system.

PGI 19.1 includes the following versions of the CUDA Toolkit:

- CUDA 9.2
- CUDA 10.0

You can let the compiler pick which version of the CUDA Toolkit to use or you can instruct it to use a particular version. The rest of this section describes all of your options.

If you do not specify a version of the CUDA Toolkit, the compiler uses the version of the CUDA Driver installed on the system on which you are compiling to determine which CUDA Toolkit to use. This auto-detect feature was introduced in the PGI 18.7 release; auto-detect is especially convenient when you are compiling and running your application on the same system. In the absence of any other information, the compiler will look for a CUDA Toolkit version in the PGI `2019/cuda` directory that matches the version of the CUDA Driver installed on the system. If a match is not found, the

compiler searches for the newest CUDA Toolkit version that is not newer than the CUDA Driver version. If there is no CUDA Driver installed, the PGI 19.1 compilers fall back to the default of CUDA 9.2.

If the only PGI compiler you have installed is PGI 19.1, then:

- If your CUDA Driver is 10.0, the compilers use CUDA Toolkit 10.0.
- If your CUDA Driver is 9.2, the compilers use CUDA Toolkit 9.2.
- If your CUDA Driver is 9.1, the compilers will issue an error that CUDA Toolkit 9.1 was not found; CUDA Toolkit 9.1 is not bundled with PGI 19.1
- If you do not have a CUDA driver installed on the compilation system, the compilers use the default CUDA Toolkit version 9.2.
- If your CUDA Driver is newer than CUDA 10.0, the compilers will still use the CUDA Toolkit 10.0. The compiler selects the newest CUDA Toolkit it finds that is not newer than the CUDA Driver.

You can change the compiler's default selection for CUDA Toolkit version using one of the following methods:

- Use a compiler option. Add the `cudaX.Y` sub-option to `-Mcuda` or `-ta=tesla` where `X.Y` denotes the CUDA version. For example, to compile a C file with the CUDA 9.2 Toolkit you would use:

  ```
  pgcc -ta=tesla:cuda9.2
  ```

  Using a compiler option changes the CUDA Toolkit version for one invocation of the compiler.
- Use an rcfile variable. Add a line defining `DEFCUDAVERSION` to the `siterc` file in the installation `bin/` directory or to a file named `.mypgirc` in your home directory. For example, to specify the CUDA 9.2 Toolkit as the default, add the following line to one of these files:

  ```
  set DEFCUDAVERSION=9.2;
  ```

  Using an rcfile variable changes the CUDA Toolkit version for all invocations of the compilers reading the rcfile.

When you specify a CUDA Toolkit version, you can additionally instruct the compiler to use a CUDA Toolkit installation different from the defaults bundled with the current PGI compilers. While most users do not need to use any other CUDA Toolkit installation than those provided with PGI, situations do arise where this capability is needed. Developers working with pre-release CUDA software may occasionally need to test with a CUDA Toolkit version not included in a PGI release. Conversely, some developers might find a need to compile with a CUDA Toolkit older than the oldest CUDA Toolkit installed with a PGI release. For these users, PGI compilers can interoperate with components from a CUDA Toolkit installed outside of the PGI installation directories.

PGI tests extensively using the co-installed versions of the CUDA Toolkits and fully supports their use. Use of CUDA Toolkit components not included with a PGI install is done with your understanding that functionality differences may exist.

To use a CUDA toolkit that is not installed with a PGI release, such as CUDA 8.0 with PGI 19.1, there are three options:

- Use the rcfile variable DEFAULT_CUDA_HOME to override the base default

```
set DEFAULT_CUDA_HOME = /opt/cuda-8.0;
```

- Set the environment variable CUDA_HOME

```
export CUDA_HOME=/opt/cuda-8.0
```

- Use the compiler compilation line assignment CUDA_HOME=

```
pgfortran CUDA_HOME=/opt/cuda-8.0
```

The PGI compilers use the following order of precedence when determining which version of the CUDA Toolkit to use.

1. If you do not tell the compiler which CUDA Toolkit version to use, the compiler picks the CUDA Toolkit from the PGI installation directory 2019/cuda that matches the version of the CUDA Driver installed on your system. If the PGI installation directory does not contain a direct match, the newest version in that directory which is not newer than the CUDA driver version is used. If there is no CUDA driver installed on your system, the compiler falls back on an internal default; in PGI 18.10, this default is CUDA 9.1.
2. The rcfile variable DEFAULT_CUDA_HOME will override the base default.
3. The environment variable CUDA_HOME will override all of the above defaults.
4. The environment variable PGI_CUDA_HOME overrides all of the above; it is available for advanced users in case they need to override an already-defined CUDA_HOME.
5. A user-specified cudaX.Y sub-option to -Mcuda and -ta=tesla will override all of the above defaults and the CUDA Toolkit located in the PGI installation directory 2019/cuda will be used.
6. The compiler compilation line assignment CUDA_HOME= will override all of the above defaults (including the cudaX.Y sub-option).

7.8. Compute Capability

The compilers can generate code for NVIDIA GPU compute capabilities 3.0 through 7.5. The compilers construct a default list of compute capabilities that matches the compute capabilities supported by the GPUs found on the system used in compilation. If there are no GPUs detected, the compilers select cc35, cc60, and cc70.

You can override the default by specifying one or more compute capabilities using either command-line options or an rcfile.

To change the default with a command-line option, provide a comma-separated list of compute capabilities to -ta=tesla: for OpenACC or -Mcuda= for CUDA Fortran.

To change the default with an rcfile, set the **DEFCOMPUTECAP** value to a blank-separated list of compute capabilities in the siterc file located in your installation's bin directory:

```
set DEFCOMPUTECAP=60 70;
```

Alternatively, if you don't have permissions to change the siterc file, you can add the **DEFCOMPUTECAP** definition to a separate .mypgirc file in your home directory.

The generation of device code can be time consuming, so you may notice an increase in compile time as the number of compute capabilities increases.

7.9. Compiling an Accelerator Program

Several compiler options are applicable specifically when working with accelerators. These options include `-ta`, `-acc`, and `-Minfo`.

7.9.1. -ta

Enable OpenACC and specify the type of accelerator to which to target accelerator regions.

-ta suboptions

There are three primary suboptions:

host
Compile OpenACC for serial execution on the host CPU; `host` has no suboptions.

multicore
Compile OpenACC for parallel execution on the host CPU; `multicore` has no suboptions.

tesla
Compile OpenACC for parallel execution on a Tesla GPU; `tesla` supports suboptions.

Multiple target accelerators can be specified. By default, the compiler generates code for `-ta=tesla,host`.

-ta=tesla suboptions

The `tesla` sub-option to `-ta` can itself be given suboptions. The following secondary suboptions are supported:

ccXY
Generate code for a device with compute capability X.Y. Multiple compute capabilities can be specified, and one version will be generated for each. By default, the compiler will detect the compute capability for each installed GPU. Use -help -ta to see the valid compute capabilities for your installation.

ccall
Generate code for all compute capabilities supported by this platform and by the selected or default CUDA Toolkit.

cudaX.Y
Use CUDA X.Y Toolkit compatibility, where installed

7.5, 8.0, 9.0, 9.1
Support for the X.Y suboption has been removed. Use the cudaX.Y suboption instead.

[no]debug
Enable [disable] debug information generation in device code

deepcopy
Enable full deep copy of aggregate data structions in OpenACC; Fortran only
fastmath
Use routines from the fast math library
[no]flushz
Enable [disable] flush-to-zero mode for floating point computations on the GPU
[no]fma
Generate [do not generate] fused multiply-add instructions; default at `-O3`
keep
Keep the kernel files (.bin, .ptx, source)
[no]lineinfo
Enable [disable] GPU line information generation
[no]nvvm
Generate [do not generate] code using the NVVM-based back-end
loadcache:{L1|L2}
Choose what hardware level cache to use for global memory loads; options include the default, `L1`, or `L2`
managed
Use CUDA Managed Memory
maxregcount:n
Specify the maximum number of registers to use on the GPU; leaving this blank indicates no limit
pinned
Use CUDA Pinned Memory
[no]rdc
Generate [do not generate] relocatable device code.
safecache
Allow variable-sized array sections in cache directives; compiler assumes they fit into CUDA shared memory
[no]unroll
Enable [disable] automatic inner loop unrolling; default at `-O3`
zeroinit
Initialize allocated device memory with zero
autocompare
Automatically compare CPU/GPU results: implies redundant
redundant
Redundant CPU/GPU execution

Usage

In the following example, tesla is the accelerator target architecture and the accelerator generates code for compute capabilities 6.0 and 7.0.

```
$ pgfortran -ta=tesla:cc60,cc70
```

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

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

DWARF Debugging Formats

PGI's debugging capability for Tesla uses the LLVM back-end. Use the compiler's `-g` option to enable the generation of full dwarf information on both the host and device; in the absence of other optimization flags, `-g` sets the optimization level to zero. If a `-O` option raises the optimization level to one or higher, only GPU line information is generated on the device even when `-g` is specified. To enforce full dwarf generation for device code at optimization levels above zero, use the `debug` sub-option to `-ta=tesla`. Conversely, to prevent the generation of dwarf information for device code, use the `nodebug` sub-option to `-ta=tesla`. Both `debug` and `nodebug` can be used independently of `-g`.

7.9.2. -acc

Enable OpenACC directives.

-acc suboptions

The following suboptions may be used:

[no]autopar
Enable [disable] loop autoparallelization within acc parallel. The default is to autoparallelize, that is, to enable loop autoparallelization.

legacy
Suppress warnings about deprecated PGI accelerator directives.

[no]routineseq
Compile every routine for the devicee. The default behavior is to not treat every routine as a seq directive.

strict
Instructs the compiler to issue warnings for non-OpenACC accelerator directives.

sync
Ignore async clauses

verystrict
Instructs the compiler to fail with an error for any non-OpenACC accelerator directive.

[no]wait
Wait for each device kernel to finish. Kernel launching is blocked by default unless the async clause is used.

Usage

The following command-line requests that OpenACC directives be enabled and that an error be issued for any non-OpenACC accelerator directive.

```
$ pgfortran -acc=verystrict -g prog.f
```

7.10. Multicore Support

PGI Accelerator OpenACC compilers support the option `-ta=multicore`, to set the target accelerator for OpenACC programs to the host multicore CPU. This will compile OpenACC compute regions for parallel execution across the cores of the host processor or processors. The host multicore will be treated as a shared-memory accelerator, so the data clauses (**`copy`**, **`copyin`**, **`copyout`**, **`create`**) will be ignored and no data copies will be executed.

By default, `-ta=multicore` will generate code that will use all the available cores of the processor. If the compute region specifies a value in the **`num_gangs`** clause, the minimum of the **`num_gangs`** value and the number of available cores will be used. At runtime, the number of cores can be limited by setting the environment variable **`ACC_NUM_CORES`** to a constant integer value. The number of cores can also be set with the **`void acc_set_num_cores(int numcores)`** runtime call. If an OpenACC compute construct appears lexically within an OpenMP parallel construct, the OpenACC compute region will generate sequential code. If an OpenACC compute region appears dynamically within an OpenMP region or another OpenACC compute region, the program may generate many more threads than there are cores, and may produce poor performance.

The ACC_BIND environment variable is set by default with `-ta=multicore`; **`ACC_BIND`** has similiar behavior to **`MP_BIND`** for OpenMP.

The `-ta=multicore` option differs from the `-ta=host` option in that `-ta=host` generates sequential code for the OpenACC compute regions.

7.11. Running an Accelerator Program

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

- When running programs on NVIDIA GPUs, the program looks for and dynamically loads the CUDA libraries.
- 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 audience. You can avoid this delay by running the `pgcudainit` program in the background, which keeps the GPU powered on.
- If you compile a program for a particular accelerator type, then run the program on a system without that accelerator, or on a system where the target libraries are

not in a directory where the runtime library can find them, the program may fail at runtime with an error message.

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

7.12. OpenACC Error Handling

The OpenACC specification provides a mechanism to allow you to intercept errors triggered during execution on a GPU and execute a specific routine in response before the program exits. For example, if an MPI process fails while allocating memory on the GPU, the application may want to call `MPI_Abort` to shut down all the other processes before the program exits. This section explains how to take advantage of this feature.

To intercept errors the application must give a callback routine to the OpenACC runtime. To provide the callback, the application calls `acc_set_error_routine` with a pointer to the callback routine.

The interface is the following, where `err_msg` contains a description of the error:

```
typedef void (*exitroutinetype)(char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);
```

When the OpenACC runtime detects a runtime error, it will invoke the `callback_routine`.

This feature is not the same as error recovery. If the callback routine returns to the application, the behavior is decidedly undefined.

Let's look at this feature in more depth using an example.

Take the MPI program below and run it with two processes. Process 0 tries to allocate a large array on the GPU, then sends a message to the second process to acknowledge the success of the operation. Process 1 waits for the acknowledgment and terminates upon receiving it.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define N 2147483648

int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int ack;
  if(rank == 0) {
    float *a = (float*) malloc(sizeof(float) * N);

#pragma acc enter data create(a[0:N])
```

```
#pragma acc parallel loop independent
    for(int i = 0; i < N; i++) {
      a[i] = i *0.5;
    }
#pragma acc exit data copyout(a[0:N])
    printf("I am process %d, I have initialized a vector of size %ld bytes on
 the GPU. Sending acknowledgment to process 1.", rank, N);
    ack = 1;
    MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if(rank == 1) {
    MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("I am process %d, I have received the acknowledgment from process 0
 that data in the GPU has been initialized.\n", rank, N);
    fflush(stdout);
  }

  // do some more work

  MPI_Finalize();

  return 0;
}
```

We compile the program with:

```
$ mpicc -ta=tesla -o error_handling_mpi error_handling_mpi.c
```

If we run this program with two MPI processes, the output will look like the following:

```
$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
call to cuMemAlloc returned error 2: Out of memory

--------------------------------------------------------
Primary job terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
--------------------------------------------------------
--------------------------------------------------------------------------
mpirun detected that one or more processes exited with non-zero status,
thus causing the job to be terminated.
```

Process 0 failed while allocating memory on the GPU and terminated unexpectedly with an error. In this case `mpirun` was able to identify that one of the processes failed, so it shut down the remaining process and terminated the application. A simple two-process program like this is straightforward to debug. In a real world application though, with hundreds or thousands of processes, having a process exit prematurely may cause the application to hang indefinitely. Therefore it would be ideal to catch the failure of a process, control the termination of the other processes, and provide a useful error message.

We can use the OpenACC error handling feature to improve the previous program and correctly terminate the application in case of failure of an MPI process.

In the following sample code, we have added an error handling callback routine that will shut down the other processes if a process encounters an error while executing on the GPU. Process 0 tries to allocate a large array into the GPU and, if the operation is successful, process 0 will send an acknowledgment to process 1. Process 0 calls the OpenACC function `acc_set_error_routine` to set the

function `handle_gpu_errors` as an error handling callback routine. This routine prints a message and calls `MPI_Abort` to shut down all the MPI processes. If process 0 successfully allocates the array on the GPU, process 1 will receive the acknowledgment. Otherwise, if process 0 fails, it will terminate itself and trigger the call to `handle_gpu_errors`. Process 1 is then terminated by the code executed in the callback routine.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define N 2147483648


typedef void (*exitroutinetype)(char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);

void handle_gpu_errors(char *err_msg) {
  printf("GPU Error: %s", err_msg);
  printf("Exiting...\n\n");
  MPI_Abort(MPI_COMM_WORLD, 1);
  exit(-1);
}


int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int ack;
  if(rank == 0) {
    float *a = (float*) malloc(sizeof(float) * N);


    acc_set_error_routine(&handle_gpu_errors);


#pragma acc enter data create(a[0:N])
#pragma acc parallel loop independent
    for(int i = 0; i < N; i++) {
      a[i] = i *0.5;
    }
#pragma acc exit data copyout(a[0:N])
    printf("I am process %d, I have initialized a vector of size %ld bytes on
 the GPU. Sending acknowledgment to process 1.", rank, N);
    fflush(stdout);
    ack = 1;
    MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if(rank == 1) {
    MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("I am process %d, I have received the acknowledgment from process 0
 that data in the GPU has been initialized.\n", rank, N);
    fflush(stdout);
  }

  // more work

  MPI_Finalize();
```

```
  return 0;
}
```

Again, we compile the program with:

```
$ mpicc -ta=tesla -o error_handling_mpi error_handling_mpi.c
```

We run the program with two MPI processes and obtain the output below:

```
$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
GPU Error: call to cuMemAlloc returned error 2: Out of memory
Exiting...

--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.
```

This time the error on the GPU was intercepted by the application which managed it with the error handling callback routine. In this case the routine printed some information about the problem and called `MPI_Abort` to terminate the remaining processes and avoid any unexpected behavior from the application.

7.13. Environment Variables

This section summarizes the environment variables that PGI OpenACC supports. These environment variables are user-setable environment variables that 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 of 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.

The following table contains the environment variables that are currently supported and provides a brief description of each.

Table 17 Supported Environment Variables

Use this environment variable...	To do this...
PGI_ACC_BUFFERSIZE	For NVIDIA CUDA devices, this defines the size of the pinned buffer used to transfer data between host and device.
PGI_ACC_CUDA_PROFSTOP	Set to 1 (or any positive value) to tell the PGI runtime environment to insert an 'atexit(cuProfilerStop)' call upon exit. This behavior may be desired in the case where a profile is incomplete or where a message is issued to call cudaProfilerStop().
PGI_ACC_DEBUG	Set to 1 to instruct the PGI runtime to generate information about device memory allocation, data movement, kernel launches, and

Use this environment variable...	To do this...
	more. PGI_ACC_DEBUG is designed mostly for use in debugging the runtime itself, but it may be helpful in understanding how the program interacts with the device. Expect copious amounts of output.
PGI_ACC_DEVICE_NUM = = ACC_DEVICE_NUM	Sets the default device number to use. PGI_ACC_DEVICE_NUM overrides 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.
PGI_ACC_DEVICE_TYPE = = ACC_DEVICE_TYPE	Sets the default device type to use. PGI_ACC_DEVICE_TYPE overrides ACC_DEVICE_TYPE. 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, TESLA, or HOST
PGI_ACC_GANGLIMIT	For NVIDIA CUDA devices, this defines the maximum number of gangs (CUDA thread blocks) that will be launched by a kernel.
PGI_ACC_NOTIFY	Writes out a line for each kernel launch and/or data movement. When set to an integer value, the value, is used as a bit mask to print information about kernel launches (value 1), data transfers (value 2), region entry/exit (value 4), wait operations or synchronizations with the device (value 8), and device memory allocates and deallocates (value 16).
PGI_ACC_PROFLIB	Enables 3rd party tools interface using the new profiler dynamic library interface.
PGI_ACC_SYNCHRONOUS	Disables asynchronous launches and data movement.
PGI_ACC_TIME	Enables a lightweight profiler to measure data movement and accelerator kernel execution time and print a summary at the end of program execution.

7.14. Profiling Accelerator Kernels

Support for Profiler/Trace Tool Interface

PGI compilers support the OpenACC Profiler/Trace Tools Interface. This is the interface used by the PGI profiler to collect performance measurements of OpenACC programs.

Using PGI_ACC_TIME

Setting the environment variable PGI_ACC_TIME to a nonzero value enables collection and printing of simple timing information about the accelerator regions and generated kernels.

Turn off all CUDA Profilers (NVIDIA's Visual Profiler, NVPROF, CUDA_PROFILE, etc) when enabling PGI_ACC_TIME, they use the same library to gather performance data and cannot be used concurently.

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.

7.15. OpenACC 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.

In Fortran, none of the OpenACC runtime library routines may be called from a PURE or ELEMENTAL procedure.

7.15.1. 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.

7.15.2. Runtime Library Routines

Table 18 lists and briefly describes the runtime library routines supported by PGI in addition to the standard OpenACC runtine API routines.

Table 18 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.

This Runtime Library Routine...	Does this...
acc_exec_time	Returns the number of microseconds spent on the accelerator executing kernels.
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.
acc_get_num_devices	Returns the number of accelerator devices of the given type attached to the host.
acc_kernels	Returns the number of accelerator kernels launched since the start of the program.
acc_present_dump	Summarizes all data present on the current device.
acc_present_dump_all	Summarizes all data present on all devices.
acc_regions	Returns the number of accelerator regions entered since the start of the program.
acc_total_time	Returns the number of microseconds spent in accelerator compute regions and in moving data for accelerator data regions.

7.16. 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.

7.16.1. Supported Fortran Intrinsics Summary Table

Table 19 is an alphabetical summary of the supported Fortran intrinsics that the accelerator supports. These functions are specific to Fortran 90/95 unless otherwise specified.

For complete descriptions of these intrinsics, refer to 'Fortran Intrinsics' of the PGI Fortran Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19fortref-openpower.pdf.

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 19 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.
COS	S,D,C,Z	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,C,Z	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,C,Z	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.

This intrinsic		Returns this value ...
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.
POW		value of the first argument raised to the power of the second argument.
REAL	I,S,D	conversion of the argument to real.
SIGN		absolute value of A times the sign of B.
SIN	S,D,C,Z	value of the sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D,C,Z	square root of the argument.
TAN		tangent of the specified value.
TANH		hyperbolic tangent of the specified value.

7.16.2. 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 20 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.
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.

This intrinsic	Returns this value ...
tan	tangent of the specified value.
tanh	hyperbolic tangent of the specified value.

Table 21 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.

Chapter 8. PCAST

PGI Compiler Assisted Software Testing (PCAST) is a set of capabilities intended to help developers test for program correctness and determine points of divergence. PCAST is useful for detecting when results differ between CPU and GPU versions of code and for testing program correctness after some modification. Such modifications can be source changes, linking against a different library, build differences (compiler flags), or even compiling for a new processor architecture.

8.1. Overview

Comparisons can be performed in two ways. The first saves the initial run's data into a file through the **pgi_compare** run-time call. The following time the program is run, the same run-time call will compare the computed results with the initial data and report the differences. The second approach handles comparisons between GPU results and CPU results. Comparisons can be invoked implicitly at the data regions with the **autocompare** flag, or explicitly with the **acc_compare** call.

With the autocompare flag, OpenACC regions will run redundantly on the CPU and GPU. On an OpenACC region exit where data is to be downloaded from device to host, PCAST will compare the values calculated on the CPU with those calculated in the GPU. Comparisons done with **autocompare** or **acc_compare** are handled in memory and do not write results to an intermediate file.

The following table outlines the supported data types that can be used with PCAST. Short, integer, and long types are not supported with **ABS**, **REL**, **ULP**, or **IEEE** options; only a bit-for-bit comparison is supported.

For floating-point types, PCAST can calculate absolute, relative, and unit-last-place differences. Absolute differences measures only the absolute value of the difference (subtraction) between two values, i.e. *abs(A-B)*. Relative differences are calculated as a ratio between the difference of values, *A-B*, and the previous value *A*; *abs((A-B)/A)*. Unit-least precision (Unit-last place) is a measure of the smallest distance between two values *A* and *B*. With the **ULP** option set, PCAST will report if the calculated ULP between two numbers is greater than some threshold.

Table 22 Supported Types for Tolerance Measurements

C/C++ Type	Fortran Type	ABS	REL	ULP	IEEE
float	real, real(4)	Yes	Yes	Yes	Yes
double	double precision, real(8)	Yes	Yes	Yes	Yes
float _Complex	complex, complex(4)	Yes	Yes	Yes	Yes
double _Complex	complex(8)	Yes	Yes	Yes	Yes
(un)signed short	integer(2)	N/A	N/A	N/A	N/A
(un)signed int	integer, integer(4)	N/A	N/A	N/A	N/A
(un)signed long	integer(8)	N/A	N/A	N/A	N/A

8.1.1. Using pgi_compare

The run-time call **pgi_compare** is designed to highlight differences between successive program runs. It has two modes of operation. The first is a data gathering phase, where the program is run normally. Calls to **pgi_compare** save specified data to a file on the disk. By default, *pgi_compare.dat* is written to in the same directory as the executable. The file name can be changed with the **PGI_COMPARE** environment variable described in the Environment Variables section.

The second can be thought of as a comparison phase. On the next run of the program, the same calls to **pgi_compare** will load previously computed values from the file (e.g. *pgi_compare.dat*) and compare those against values in program memory. Subsequent runs will continue to use the data in the file as a golden copy so long as the file exists. If the file does not exist, then **pgi_compare** will assume it is in the data gathering phase and create the file.

The signature of **pgi_compare** is as follows:

```
size_t pgi_compare(const void *hostptr, \
    const char *dtypename, \
    unsigned long count, \
    const char *varname, \
    const char *filename, \
    const char *funcname, \
    int linenum);
```

The first argument, **hostptr**, is the address of the data to be saved and compared against. It is followed by a string that describes the data type, **dtypename**, and the number of elements to compare, **count**. The remaining arguments: strings **varname**, **filename**, **funcname**, and integer **linenum** respectively are identifiers in the report output. The caller should give meaningful names to the last four arguments. They can be anything, since they only serve to annotate the report. It is imperative that the identifiers are not modified between comparisons; comparisons must be called in the same order for each program run. If, for example, you are calling **pgi_compare** inside a loop, it is reasonable to set the last argument to be the loop index.

Consider the following example in which we would like to compare the results of some solver. Add a few **pgi_compare** calls after invoking the solve routine to mark which variables should be compared.

```
void solve(double* a, double* b, double* r, int* pivot, int n){
    int fail;
    dgesv(n, 1, a, n, pivot, b, n, &fail);
}
....
solve(a, b, r, pivot, n);
pgi_compare(b, "double", n, "b", "my-file-name", "my-solver-func", 1);
pgi_compare(a, "double", n*n, "a", "my-file-name", "my-solver-func", 2);
pgi_compare(pivot, "int", n, "pivot", "my-file-name", "my-solver-func", 3);
```

Running the code with **PGI_COMPARE=create** will create a comparison file that holds the contents of *b*, *a*, and *pivot* into a file. This data serves as the baseline that future runs are compared against.

```
$ pgcc example.c -o example -ta=tesla
$ PGI_COMPARE=create,datafile=comparison.dat ./example
```

After the initial run, change the dgesv call to a different solver. Recompile and run the program with **PGI_COMPARE=compare**. Set any desired comparison options like **abs** or **rel** in the environment variable. Results from the new solver will be compared against the baseline previously gathered.

```
$ PGI_COMPARE=compare,summary,abs=1,datafile=comparison.dat ./example
```

Take note that since the reference data is saved to a file, you can perform many comparisons without having to regenerate the reference; it only needs to be collected once and can compared against multiple times. You will want to take caution when placing **pgi_compare** calls, especially inside loops. Data can grow at a very fast rate depending on the amount of data compared and how frequently it is called. It is a good idea to use **pgi_compare** sparingly or on programs where the total data size is relatively small.

8.1.2. Using acc_compare

To compare results against the CPU and GPU, you can use **acc_compare** to explicitly compare results between host and device. To use **acc_compare**, you must compile with **-ta=tesla:redundant**. The compiler flag forces both the CPU and GPU to compute results redundantly. At the **acc_compare** call, the values in GPU memory are compared against the same computations in CPU memory.

acc_compare takes in a pointer to the data to be compared, *hostptr*, and the number of elements to compare, *count*. The type can be inferred at run-time, so it is only necessary to specify the number of items to compare, not the total byte length.

```
void acc_compare(void *hostptr, unsigned long count);
```

You can call **acc_compare** on any variable or array that is present in device memory. You can also call **acc_compare_all()** (no arguments) to compare all values that are present in device memory against the correponding values in host memory.

Consider the following example using **acc_compare**. At the end of each parallel section, we call **acc_compare**. At this point, both the CPU and GPU hold the results of *a2* (since the code was executed redundantly on both processors). A total of five comparisons are done in-memory at the end of each *t* iteration.

```
#pragma acc data copy(a1[0:size], a2[0:size])
  {
    for (t = 0; t < 5; t++) {
      #pragma acc parallel
      for(i = 0; i < size; i++) {
 a2[i] += a1[i];
      }
      acc_compare(a2, size);
    }
  }
```

Comparisons are done in-memory; there is no external file that is read/written to. Similar to the **pgi_compare** call, control parameters are set before invocation in the **PGI_COMPARE** environment variable.

```
$ pgcc -o example example.c -ta=tesla:redundant
$ PGI_COMPARE=summary,abs=1 ./example
```

8.1.3. Using autocompare

Instead of explicitly telling the compiler when to copy memory between the CPU and GPU, the **-ta=tesla:autocompare** flag will automatically do comparisons between CPU and GPU memory on OpenACC data exit regions; essentially comparisons are done whenever memory would be copied off the GPU and onto the CPU. The autocompare flag implies **-ta=tesla:redundant**, since both types of processors need to compute the results for the comparison.

Autocompare is the simplest way to use PCAST, as it only requires adding a flag to the compilation step. However, it will run comparisons on all OpenACC data regions. It is invoked whenever data would normally be copied off the GPU and back into CPU memory (i.e during a copy, copyout, or update host directive).

Consider the following matrix vector product routine:

```
void matvec(double* a, double* x, double* v, int n)
{
  #pragma acc parallel loop \
  copyin(a[0:n*n], x[0:n]) copyout(v[0:n])
  for (int i = 0; i < n; ++i)
  {
    double r = 0.0;
    #pragma acc loop reduction(+:r)
    for (int j = 0; j < n; ++i)
    {
      r += a[i*n+j] * x[j];
    }
    v[i] = r;
  }
}
```

If we build for GPU acceleration without using **-ta=tesla:autocompare**, the code will run through the following sequence:

1. Allocate space for *a*, *x*, and *v*
2. Copy *a* and *x* from the host to the device
3. Launch the compute kernel on the device for the reduction
4. Copy *v* from the device back into host memory
5. Deallocate *a*, *x*, and *v* on the device

If we add the autocompare flag in, the sequence of operations would look like so:

1. Allocate space for *a*, *x*, and *v*
2. Copy *a* and *x* from the host to the device
3. Launch the compute kernel on the device for the reduction
4. Compute the same reduction on the CPU
5. Copy *v* (which holds the GPU results) from the device back into host memory
6. Compare the GPU calculated value of *v* against the same computation done in the CPU and report the differences
7. Deallocate *a*, *x*, and *v* on the device

Using the autocompare flag yields a more comprehensive coverage of your program, but it can be wasteful if you are only interested in comparisons within a particular region. We suggest using the autocompare flag first, as an intial triage, and later inspect the problematic sections specifically with the aforementioned run-time methods.

8.2. Limitations

There are currently a few limitations with using PCAST that are worth keeping in mind.

- Comparisons are not thread-safe. If you are using PCAST with multiple threads, ensure that only one thread is doing the comparisons. This is especially true if you are using PCAST with MPI. If you use **pgi_compare** with MPI, you must make sure that only one thread is writing to the comparison file. Or, use a script to set PGI_COMPARE to encode the file name with the MPI rank.
- Comparisons must be done with like types; you cannot compare one type with another. It is not possible to, for example, check for differing results after changing from double precision to single. Comparisons are limited to those present in table Table 22. Currently there is no support for structured or derived types.
- The **-ta=tesla:managed** option is incompatible with autocompre and **acc_compare**. Both the CPU and GPU need to calculate result separately and to do so they must have their own working memory spaces.
- If you do any data movement on the device, you must account for it on the host. For example, if you are using CUDA-aware MPI or GPU-accelerated libraries that modify device data, then you must also make the host aware of the changes. In these cases it is helpful to use the **host_data** clause, which allows you to use device addresses within host code.

- Currently we do not support comparisons in the form of directives, e.g. **`#pragma acc compare(...)`** or **`#pragma pgi compare(...)`**. Support for comparisons through directives will be added in a future release. With directives, all comparisons can be enabled or disabled with a switch at compile-time. The same functionality can be achieved by surrounding the run-time calls with preprocessor macro blocks, though this is cumbersome and bulkier.

8.3. Environment Variables

Behavior of PCAST/Autocompare is controlled through the **`PGI_COMPARE`** variable. Options can be specified in a comma-separated list: **`PGI_COMPARE=<opt1>,<opt2>,...`**

If no options are specified, the default is to perform comparisons with *abs=0*. Comparison options are not mutually exclusive. PCAST can compare absolute differences with some *n=3* and relative differences with a different threshold, e.g. *n=5*; *PGI_COMPARE=abs=3,rel=5,....*

You can specify either an absolute or relative location to be used with the datafile option. The parent directory should be owned by the same user executing the comparisons and the datafile should have the appropriate read/write permissions set.

Table 23 PGI_COMPARE Options

Option	Description	Minimum supported version
abs=n	Compare absolute difference; tolerate differences up to 10^(-n). Default value is 0	18.7
create[1]	Specifies that this is the run that will produce the reference results	18.7
compare[1]	Specifies that the current run will be compared with a reference file	18.7
datafile="name"[1]	Name of the file that data will be saved to in binary format to avoid loss of precision. If empty will use the default, 'pgi_compare.dat'	18.7
ieee	Compare IEEE NaN checks	18.7
outputfile="name"	File to write comparison results to. If empty, comparison report will be sent to stderr	18.7
patch	Patch errors (outside tolerance) with correct values	18.10
patchall	Patch all differences (inside and outside tolerance) with correct values	18.10
rel=n	Compare relative difference; tolerated differences up to 10^(-n). Default value is 0	18.7

[1] Only available for **`pgi_compare`**

Option	Description	Minimum supported version
report=n	Report up to n (default of 50) passes and/or fails; without reportpass will only report failures, with reportpass will report passes and fails	18.7[2]
reportall	Report all passes and fails (overrides limit set in report=n)	18.10
reportpass	Report passes; respects limit set with report=n	18.10
silent	Suppress output - overrides all other output options, including summary and verbose	18.10
stop	Stop at first differences	18.7
summary	Print summary of comparisons at end of run	18.7
ulp=n	Compare Unit of Least Precision difference	18.7
verbose	Outputs more details of comparison (including patches)	18.10
verboseautocompare	Outputs verbose reporting of what and where the host is comparing (autocompare only)	18.10

[2] Behavior modified in 18.10

Chapter 9. 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.

9.1. Install Eclipse CDT

To install the Eclipse plugin for the PGI C and C++ compilers:

1. Download the Eclipse plugin from the PGI download webpage, pgicompilers.com/downloads .
2. Untar and install the PGI Eclipse plugin componennts.

   ```
   $ tar zxf pgieclipse-2019.tar.gz
   $ ./install
   ```

   Note you will be asked to provide the location of your current PGI installation. The default is `/opt/pgi`.
3. Before you install the actual PGI Eclipse plugin, from within Eclipse, 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.
4. Go to Help -> Install New software.
5. Click the Add button to add a new software repository.
6. In the Add Repository dialog box:

   1. Click Local.
   2. Select your PGI installation directory, such as `/opt/pgi`.
   3. Browse inside `2019/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>/2019/eclipse/cdt7`; the CDT 8 plugin is in `/opt/pgi/<os-version>/2019/eclipse/cdt8`, and so on.

5. Click OK in the Add Repository dialog.

The install form now shows "PGI C/C++ Compiler Plugin" as an option to install.

7. Check the box next to PGI option and select Next to get to the Install Details view.
8. Click Next again.
9. Review and accept the End-User License agreement.
10. Click Finish.

You are prompted to restart. Select Restart to complete installation of the plugin.

9.2. Use Eclipse CDT

To use the Eclipse plugin for the PGI C and C++ compilers, the directory containing PGI compilers and tools should be included in your PATH *prior* to starting the Eclipse IDE. For details on how to include this directory in your PATH environment variable, refer to Using Environment Variables, and specifically to PATH.

This plugin does not currently support the Code Analysis feature of Eclipse CDT. This feature is not disabled by default for PGI projects and is the cause of spurious syntax errors during pre-compilation. This feature can be disabled manually at either the Project or Workspace level; any actual compilation or link errors are reported at build time.

- To disable Code Analysis at the Workspace level, select the menu item Window | Preferences, then select C/C++, then Code Analysis; uncheck all category items, which should deselect all sub-items.
- To disable Code Analysis at the Project level, select the menu item Project | Properties, then select C/C++ General, then Code Analysis; uncheck the top-level categories to deselect everything.

The PGI plugin follows 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 10. 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.

10.1. 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
```

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$`.

10.2. 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.

10.3. 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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

- 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.

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.

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 24 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
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
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

Directive or pragma	Functionality	Default	Fortran Scope	C/C++ Scope
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

10.4. 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.

```
!pgi$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)
!pgi$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.

10.5. 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.

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.

10.6. 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.

10.6.1. Prefetch Directive Syntax in Fortran

The syntax of a prefetch directive is as follows:

```
  !$mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

10.6.2. Prefetch Directive Format Requirements

The sentinel for prefetch directives is `!$mem`, which is distinct from the `!pgi$` sentinel used for optimization directives. Any prefetch directives that use the `!pgi$` sentinel are ignored by the PGI compilers.

- The "c" must be in column 1 for fixed format.
- Either * or ! is allowed in place of c for fixed format.
- The scope indicators g, r and l used with the `!pgi$` 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.

10.6.3. Sample Usage of Prefetch Directive

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
!$mem prefetch arow(1),b(1,j)
!$mem prefetch arow(5),b(5,j)
!$mem prefetch arow(9),b(9,j)
 do k = 1, n, 4
!$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.

10.6.4. Prefetch Pragma Syntax in C/C++

The syntax of a prefetch pragma is as follows:

```
#pragma mem prefetch <var1>[,<var2>[,...]]
```

where <*varn*> is any valid variable, member, or array element reference.

10.6.5. Sample Usage of Prefetch Pragma

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];
}
```

10.7. !$PRAGMA C

When programs are compiled using one of the PGI Fortran compilers on Linux 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 !$PRAGMA C in the Fortran program to call a C/C++ function from Fortran. The statement would look similar to this:

```
!$PRAGMA C(name[,name]...)
```

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.

10.8. 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.

10.8.1. 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.

10.8.2. 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 appear in the body of an interface block or in the body of a module procedure, 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.

10.8.3. 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 25 indicates which rules are ignored for which dummy arguments in the preceding sample subroutine fragment:

Table 25 IGNORE_TKR Example

Dummy Argument	Ignored Rules
A	Type, Kind and Rank
B	Only rank
C	Type and Kind
D	Only Kind

Notice that no letters were specified for A, so all type, kind, and rank rules are ignored.

Chapter 11. CREATING AND USING LIBRARIES

A library is a collection of functions or subprograms that are grouped for reference and ease of linking. This section 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.

This section does not duplicate material related to using libraries for inlining, described in Creating an Inline Library or information related to runtime library routines available to OpenMP programmers, described in Runtime Library Routines.

PGI provides libraries that export C interfaces by using Fortran modules.

This section 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`

11.1. 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 3.5.

```
#include <math.h>
#include<stdio.h>
#define PI 3.1415926535
void main()
{
  double x, y;
  x = PI/3.0;
  y = acos(0.5);
  printf('%f %f\n',x,y);
}
```

Including `math.h` causes PGI 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	

11.2. Using System Library Routines

Release 19.1 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	

11.3. 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.

11.3.1. 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
```

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. If you have placed `tobeshared.so` in 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
   ```

   

   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.

11.3.2. 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.

11.4. Using LIB3F

The PGI Fortran compilers include complete support for the de facto standard LIB3F library routines. See the PGI Fortran Language Reference manual for a complete list of available routines in the PGI implementation of LIB3F.

11.5. LAPACK, BLAS and FFTs

All PGI products now include a BLAS and LAPACK library based on the customized OpenBLAS project source and built with PGI compilers. The LAPACK library is called liblapack.a. The BLAS library is called libblas.a. These libraries are installed to $PGI/linuxpower/19.1/lib.

To use these libraries, simply link them in using the -l option when linking your main program:

```
% pgfortran myprog.f -llapack -lblas
```

11.6. Linking with ScaLAPACK

The ScaLAPACK libraries are automatically installed with each MPI library version which accompanies a PGI installation. You can link with the ScaLAPACK libraries by specifying -Mscalapack on any of the *PGI* compiler command lines. For example:

```
% mpif90 myprog.f -Mscalapack
```

A pre-built version of the BLAS library is automatically added when the -Mscalapack switch is specified. If you wish to use a different BLAS library, and still use the -Mscalapack switch, then you can list the set of libraries explicitly on your link line. Alternately, you can copy your BLAS library into $PGI/linuxpower/19.1/lib/libblas.a.

11.7. The C++ Standard Template Library

On Linux, the GNU-compatible pgc++ compiler uses the GNU g++ header files and Standard Template Library (STL) directly. The versions used are dependent on the version of the GNU compilers installed on your system, or specified when makelocalrc was run during installation of the PGI compilers.

Chapter 12. 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 section 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 PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf and the Profiler User's Guide, www.pgroup.com/resources/docs/19.1/pdf/pgi19profug.pdf.

- 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 OpenMP section: Environment Variables.
- You can use environment variables to control the behavior of the PGI profiler. For a description of environment variables that affect this tool, refer to the Profiler User's Guide, www.pgroup.com/resources/docs/19.1/pdf/pgi19profug.pdf.

12.1. 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, let's look at how a user might initialize the shell environment prior to using the PGI compilers and tools.

12.1.1. 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/linuxpower/19.1/man
% set path = ($PGI/linuxpower/19.1/bin/ $path)
```

In bash, sh, zsh, or ksh, use these commands:

```
$ PGI=/opt/pgi; export PGI
$ MANPATH=$MANPATH:$PGI/linuxpower/19.1/man; export MANPATH
$ PATH=$PGI/linuxpower/19.1/bin:$PATH; export PATH
```

12.2. 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 15 and to the complete descriptions in 'OpenMP Environment Variables' in the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

Table 26 PGI-Related Environment Variable Summary

Environment Variable	Description
FORTRANOPT	Allows the user to specify that the PGI Fortran compilers user VAX I/O or other custom 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.
MANPATH	Sets the directories that are searched for manual pages associated with the command that the user types.
MP_WARN	Allows you to eliminate certain default warning messages.
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.
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.

Environment Variable	Description
PGI	Specifies, at compile-time, the root directory where the PGI compilers and tools are installed.
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.

12.3. PGI Environment Variables

You use the environment variables listed in Table 26 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.

12.3.1. 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.
- If `FORTRANOPT` exists and contains the value `no_minus_zero`, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) equal to negative zero will be output as if it were positive zero.
- 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
```

12.3.2. 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":"./"
```

12.3.3. 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 the linuxpower version of the compilers and tool s and allows the user access to the manual pages associated with them.

```
% set path = (/opt/pgi/linuxpower/19.1/bin $path
% setenv MANPATH "$MANPATH":/opt/pgi/linuxpower/19.1/man
```

12.3.4. 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.

12.3.5. 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 linuxpower version of the compilers and tools, or to target linuxpower as the default.

The following csh example targets linuxpower version of the compilers and tools.

```
% set path = (/opt/pgi/linuxpower/19.1/bin $path)
```

12.3.6. 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.

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 linuxpower versions of the PGI compilers and tools to be the default, you would use this syntax:

```
% setenv PGI /opt/pgi
% setenv MANPATH "$MANPATH":$PGI/linuxpower/19.1/man
% set path = ($PGI/linuxpower/19.1/bin $path)
```

12.3.7. 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.

There is a performance penalty associated with the stack cleanup.

12.3.8. 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.

12.3.9. PWD

The PWD variable allows you to display the current directory.

12.3.10. 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.

12.3.11. 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.

12.3.12. 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.

12.4. 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/linuxpower/19.1/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 pgi64/19.1
% module load pgi/19.1
```

where "`pgi/19.1`" also uses the 64-bit compilers.

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
CC	Full path to `pgcc`
CPP	Full path to `pgprepro`
CXX	Path to `pgc++`
FC	Full path to `pgfortran`
LD_LIBRARY_PATH	Prepends the PGI library directory
MANPATH	Prepends the PGI man page directory
PATH	Prepends the PGI compiler and tools `bin` directory
PGI	The base installation directory

PGI does not provide support for the Environment Modules package. For more information about the package, go to: http://modules.sourceforge.net.

Chapter 13. 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 section 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.

13.1. Deploying Applications on Linux

To successfully deploy your application on Linux, some of the issues to consider include:

- Runtime Libraries
- 64-bit Linux Systems
- Redistribution of Files
- Licensing

13.1.1. 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 version `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.

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
```

13.1.2. 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`; these directories contain the redistributed shared object libraries. These may be redistributed by licensed PGI customers under the terms of the End-User License Agreement.

13.1.3. 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, an area of significant change between some versions of Linux is in `libpthread.so` and `libnuma.so`. PGI compilers use these dynamically linked libraries for the options `-Mconcur` (auto-parallel), `-mp` (OpenMP), and `-acc -ta=multicore` (OpenACC). Statically linking these libraries may not be possible, or may result in failure at execution.

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 RHEL 7.2 should not expect the program to run without incident on a RHEL 5.2 system, an earlier Linux version. It *may* run, but it is less likely. Developers might consider building applications on earlier Linux versions for wider usage. Dynamic linking of Linux and gcc system routines on the platform executing the program can also reduce problems.

13.1.4. 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.

13.2. PGI Redistributables

PGI redistributable directories contain all of the PGI Linux runtime library shared object files that can be re-distributed by PGI 19.1 licensees under the terms of the PGI End-User License Agreement (EULA).

Chapter 14. INTER-LANGUAGE CALLING

This section describes inter-language calling conventions for C, C++, and Fortran programs using the PGI compilers. Fortran 2003 ISO_C_Binding provides a mechanism to support the interoperability with C. This includes the ISO_C_Binding intrinsic module, binding labels, and the BIND attribute. In the absence of this mechanism, 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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

This section 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' section of the PGI Compiler Reference Manual, www.pgroup.com/resources/docs/19.1/pdf/pgi19ref-openpower.pdf.

`-c` `-Mnomain` `-Miface` `-Mupcase`

14.1. Overview of Calling Conventions

This section 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

The sections Inter-language Calling Considerations through Example – C++ Calling Fortran describe how to perform inter-language calling.

14.2. 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 C89; however, it is still possible to provide inter-language calls but there are no general calling conventions for such cases.

- 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.

14.3. 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 27, Fortran and C/C++ Data Type Compatibility, lists compatible types. If the

call is to a Fortran subroutine or a Fortran `CHARACTER` 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.

14.4. Upper and Lower Case Conventions, Underscores

By default, 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, 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 !$PRAGMA C.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

14.5. Compatible Data Types

Table 27 shows compatible data types between Fortran and C/C++. Table 28, Fortran and C/C++ Representation of the COMPLEX Type 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 27 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

Fortran Type (lower case)	C/C++ Type	Size (bytes)
integer x	int x	4
integer*1 x	signed char x	1
integer*2 x	short x	2
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 28 Fortran and C/C++ Representation of the `COMPLEX` Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	float complex x;	8
complex*8 x	float complex x;	8
double complex x	double complex x;	16
complex *16 x	double complex x;	16

For C/C++, the `complex` type implies C99 or later.

14.5.1. 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.

14.6. 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.

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.

14.6.1. 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))
```

14.6.2. Character Return Values

Functions and Subroutines 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

The following example illustrates the extra parameters, `tmp` and `10`, supplied by the caller:

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.

The value of the character function is not automatically NULL-terminated.

14.7. Array Indices

C/C++ arrays and Fortran arrays use different default initial array index values. By default, arrays in C/C++ start at 0 and arrqays in Fortran 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.

14.8. Examples

This section contains examples that illustrate inter-language calling.

14.8.1. Example – Fortran Calling C

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.

C function f2c_func_ shows a C function that is called by the Fortran main program shown in 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 "_".

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
```

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
```

14.8.2. Example – C Calling Fortran

The example C Main Program c2f_main.c shows a C main program that calls the Fortran subroutine shown in 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 "_".

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);
}
```

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
```

14.8.3. Example – C++ Calling C

C++ Main Program cp2c_main.C Calling a C Function shows a C++ main program that calls the C function shown in Simple C Function c2cp_func.c.

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;
 }
```

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
$ pgc++ cp2c_main.C cp2c_func.o
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbfffffb94
func: a = 8 b = 2 ptr c = bfffffb94
func: res = 4
main: res = 4
```

14.8.4. Example – C Calling C ++

The example in C Main Program c2cp_main.c Calling a C++ Function shows a C main program that calls the C++ function shown in Simple C++ Function c2cp_func.C with Extern C.

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);
 }
```

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
$ pgc++ c2cp_main.o c2cp_func.C
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbfffffb94
func: a = 8 b = 2 ptr c = bfffffb94
func: res = 4
main: res = 4
```

You cannot use the extern "C" form of declaration for an object's member functions.

14.8.5. Example – Fortran Calling C++

The Fortran main program shown in Fortran Main Program f2cp_main.f calling a C++ function calls the C++ function shown in 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 "_":

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
```

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:

```
$ pgc++ -c f2cp_func.C
$ pgfortran f2cp_func.o f2cp_main.f -pgc++libs
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

14.8.6. Example – C++ Calling Fortran

Fortran Subroutine cp2f_func.f shows a Fortran subroutine called by the C++ main program shown in 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 "_":

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;
}
```

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
$ pgc++ 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

..

You must explicitly link in the PGFORTRAN runtime support libraries when linking pgfortran-compiled program units into C or C++ main programs.

Chapter 15. CONTACT INFORMATION

You can contact NVIDIA's PGI compilers and tools team at:

9030 NE Walker Road, Suite 100
Hillsboro, OR 97006

Or electronically using any of the following means:

Fax: +1-503-682-2637
Sales: sales@pgroup.com
WWW: https://www.pgroup.com or pgicompilers.com

The PGI User Forum, pgicompilers.com/userforum is monitored by members of the PGI engineering and support teams as well as other PGI customers. The forums contain answers to many commonly asked questions. Log in to the PGI website, pgicompilers.com/login to access the forums.

Many questions and problems can be resolved by following instructions and the information available in the PGI frequently asked questions (FAQ), pgicompilers.com/faq.

Submit support requests using the PGI Technical Support Request form, pgicompilers.com/support-request.