PGI® COMPILERS & TOOLS

PGI VISUAL FORTRAN USER'S GUIDE

Version 2018

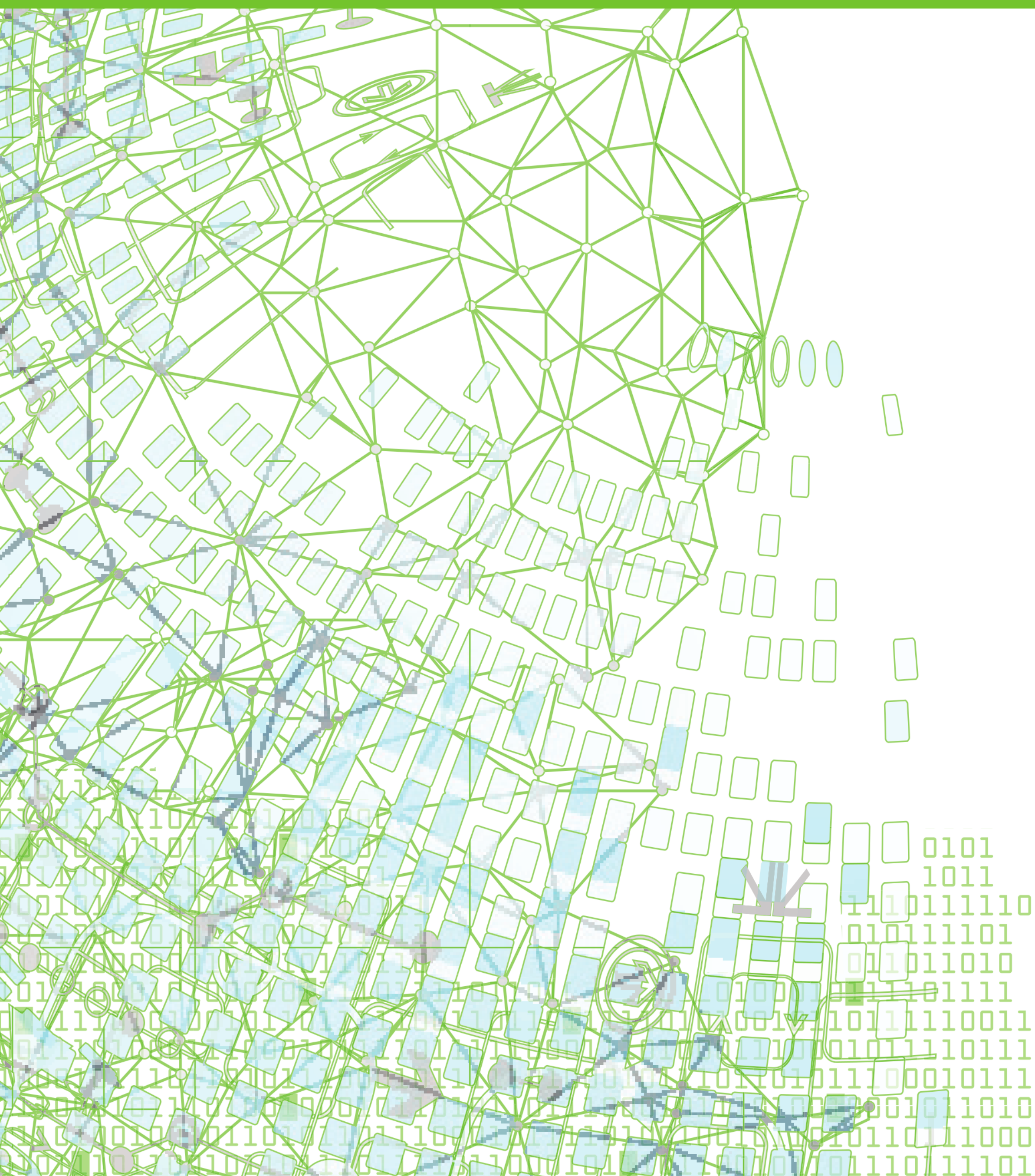

TABLE OF CONTENTS

LIST OF TABLES

PREFACE

This guide is part of a set of manuals that describe how to use the PGI Fortran compilers and program development tools integrated with Microsoft Visual Studio. These tools, combined with Visual Studio and assorted libraries, are collectively known as PGI Visual Fortran®, or PVF®. You can use PVF to edit, compile, debug, optimize, and profile serial and parallel applications for x86-64 processor-based systems.

The *PGI Visual Fortran User's Guide* provides operating instructions for both the Visual Studio integrated development environment as well as command-level compilation. The PGI Visual Fortran Reference Manual contains general information about PGI's implementation of the Fortran language. This guide does not teach the Fortran programming language.

Audience Description

This manual is intended for scientists and engineers using PGI Visual Fortran. To fully understand this guide, you should be aware of the role of high-level languages, such as Fortran, in the software development process; and you should have some level of understanding of programming. PGI Visual Fortran is available on a variety of x86-64/x64 hardware platforms and variants of the Windows operating system. You need to be familiar with the basic commands available on your system.

Compatibility and Conformance to Standards

Your system needs to be running a properly installed and configured version of this PGI product. For information on installing PVF, 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).
- 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 with PVF gives an overview of the Visual Studio environment and how to use PGI Visual Fortran in that environment.

Build with PVF gives an overview of how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment) to create and build a PVF project.

Debug with PVF gives an overview of how to use the custom debug engine that provides the language-specific debugging capability required for Fortran.

Using MPI in PVF describes how to use MPI with PGI Visual Fortran.

Getting Started with The Command Line Compilers 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 shows examples of their use.

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

Using Directives provides a description of each Fortran optimization directive, 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.

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

Hardware and Software Constraints

This guide describes versions of the PGI Visual Fortran that are intended for use on x64 processor-based systems. Details concerning environment-specific values and defaults and system-specific features or limitations are presented in the release notes delivered with the PGI Visual Fortran.

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=medium	static linking
AVX	host	-mcmodel=small	Win32
CUDA	hyperthreading (HT)	MPI	Win64
device	large arrays	multicore	x64
DLL	license keys	NUMA	s86
driver	LLVM	SIMD	x87
DWARF	manycore	SSE	

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
PGF77	ANSI FORTRAN 77	pgf77

Compiler or Tool	Language or Function	Command
PGFORTRAN	ISO/ANSI Fortran 2003	pgfortran
PGI Debugger	Source code debugger	pgdbg
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 the *PGF77* compiler, whose features are a subset of *PGFORTRAN*, is similar.

There are a wide variety of 64-bit x86-compatible processors in use. All are supported by the PGI compilers and tools. Most of these processors are forward-compatible, but not backward-compatible, meaning that code compiled to target a given processor will not necessarily execute correctly on a previous-generation processor.

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

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

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

> 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 x86-64 and x64 architectures, and the compilers and tools available from The Portland Group.

- PGI Fortran Reference Manual, www.pgroup.com/resources/docs/18.5/pdf/pgi18fortref-x86.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.).
- *System V Application Binary Interface X86-64 Architecture Processor Supplement*, http://www.x86-64.org/documentation_folder/abi.pdf.
- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *Programming in VAX Fortran, Version 4.0*, Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.

Chapter 1. GETTING STARTED WITH PVF

This section describes how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment). For information on general use of Visual Studio, refer to Microsoft's documentation.

PVF is integrated with Microsoft Visual Studio 2015. Throughout this document, "PGI Visual Fortran" refers to PVF integrated with this version of Visual Studio. Similarly, "Microsoft Visual Studio" refers to Visual Studio 2015. When it is necessary to distinguish further, the document does so.

Single-user node-locked and multi-user network floating license options are available for both products. When a node-locked license is used, one user at a time can use PVF on the single system where it is installed. When a network floating license is used, a system is selected as the server and it controls the licensing, and users from any of the client machines connected to the license server can use PVF. Thus multiple users can simultaneously use PVF, up to the maximum number of users allowed by the license.

PVF provides a complete Fortran development environment fully integrated with Microsoft Visual Studio. It includes a custom Fortran Build Engine that automatically derives build dependencies, Fortran extensions to the Visual Studio editor, a custom PGI Debug Engine integrated with the Visual Studio debugger, PGI Fortran compilers, and PVF-specific property pages to control the configuration of all of these.

The following sections provide a general overview of the many features and capabilities available to you once you have installed PVF. Exploring the menus and trying the sample program in this section provide a quick way to get started using PVF.

1.1. PVF on the Start Screen and Start Menu

PGI creates an entry on the Start Menu for PGI Visual Fortran to facilitate access to PVF, command shells pre-configured with the PVF environment, and documentation. Microsoft has replaced the Start Menu in the Windows 8.1, 10 and Server 2012 operating systems with a Start Screen. If you are using one of these environments, you find tiles on the Start Screen for Visual Studio, the PGI profiler and the command shells. The document links are hidden tiles; to locate one, search for it from the Start Screen by typing the first letter or two of its name. Tip: almost all of the PGI documents start with the letter ‘p’.

This section provides a quick overview of the PVF menu selections. To access the PGI Visual Fortran menu, from the Start menu, select *Start | All Programs | PGI Visual Fortran*.

1.1.1. Shortcuts to Launch PVF

PVF runs within Visual Studio, so to invoke PVF you must invoke Visual Studio. If other languages such as Visual C++ or Visual Basic are installed, they are available in the same instance of Visual Studio as PVF.

The PVF shortcuts include the following:

PGI Visual Fortran 2015—Select this option to invoke PGI Visual Fortran 2015.

1.1.2. Commands Submenu

From the Commands menu, you have access to PVF command shells configured for each version of Visual Studio PVF supports.

These shortcuts invoke a command shell with the environment configured for the PGI compilers and tools. The command line compilers and graphical tools may be invoked from any of these command shells without any further configuration.

Important If you invoke a generic Command Prompt using *Start | All Programs | Accessories | Command Prompt*, then the environment is not pre-configured for PGI products.

1.1.3. Profiler Submenu

Use the profiler menu to launch the PGI performance profiler. The profiler provides a way to visualize and diagnose the performance of the components of your program and provides features for helping you to understand why certain parts of your program have high execution times.

1.1.4. Documentation Menu

All PGI documentation is available online. The Documentation menu contains a link to the online location.

1.1.5. Licensing Submenu

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

- *Generate License*—Select this option to display the PGI License Setup dialog that walks you through the steps required to download and install a license for PVF. To complete this process you need an internet connection.

- *License Agreement*—Select this option to display the license agreement that is associated with use of PGI software.

1.2. Introduction to PVF

This section provides an introduction to PGI Visual Fortran as well as basic information about how things work in Visual Studio. It contains an example of how to create a PVF project that builds a simple application, along with the information on how to run and debug this application from within PVF. If you're already familiar with PVF or are comfortable with project creation in VS, you may want to skip ahead to the next section.

1.2.1. Visual Studio Settings

PVF projects and settings are available as with any other language. The first time Visual Studio is started it may display a list of default settings from which to choose; select *General Development Settings*. If Visual Studio was installed prior to the PVF install, it will start as usual after PVF is installed, except PVF projects and settings will be available.

1.2.2. Solutions and Projects

The Visual Studio IDE frequently uses the terms solution and project. For consistency of terminology, it is useful to discuss these here.

solution
: All the things you need to build your application, including source code, configuration settings, and build rules. You can see the graphical representation of your solution in the Solution Explorer window in the VS IDE.

project
: Every solution contains one or more projects. Each project produces one output, where an output is an executable, a static library, or a dynamic-link library (DLL). Each project is specific to a single programming language, like PGI Visual Fortran or Microsoft Visual C++, but you can have projects of different languages in the same solution.

We examine the relationship between a solution and its projects in more detail by using an example. But first let's look at an overview of the process. Typically there are these steps:

1. Create a new PVF project.
2. Modify the source.
3. Build the solution.
4. Run the application.
5. Debug the application.

1.3. Creating a Hello World Project

Let's walk through how to create a PVF solution for a simple program that prints "Hello World".

1. Create Hello World Project

   Follow these steps to create a PVF project to run "Hello World".

   1. Select *File* | *New* | *Project* from the Visual Studio main menu.

      The *New Project* dialog appears.
   2. In the *Project types* window located in the left pane of the dialog box, expand *PGI Visual Fortran*, and then select *x64*.
   3. In the *Templates* window located in the right pane of the dialog box, select *Console Application (64-bit)*.
   4. In the *Name* field located at the bottom of the dialog box, type: `HelloWorld`.
   5. Click OK.

   You should see the Solution Explorer window in PVF. If not, you can open it now using *View* | *Solution Explorer* from the main menu. In this window you should see a solution named HelloWorld that contains a PVF project, which is also named HelloWorld.
2. Modify the Hello World Source

   The project contains a single source file called `ConsoleApp.f90`. If the source file is not already opened in the editor, open it by double-clicking the file name in the Solution Explorer. The source code in this file should look similar to this:

   ```
   program prog
   implicit none
   ! Variables
   ! Body
   end program prog
   ```

   Now add a print statement to the body of the main program so this application produces output. For example, the new program may look similar to this:

   ```
   program prog
   implicit none
   ! Variables
   ! Body
   print *, "Hello World"
   end program prog
   ```
3. Build the Solution

   You are now ready to build a solution. To do this, from the main menu, select *Build* | *Build Solution*.

   The *View* | *Output* window shows the results of the build.
4. Run the Application

   To run the application, select *Debug* | *Start Without Debugging*.

   This action launches a command window in which you see the output of the program. It looks similar to this:

   ```
   Hello World
   Press any key to continue . . .
   ```
5. View the Solution, Project, and Source File Properties

The solution, projects, and source files that make up your application have properties associated with them.

> The set of property pages and properties may vary depending on whether you are looking at a solution, a project, or a file. For a description of the property pages that PVF supports, refer to the 'PVF Properties' section in the PGI Visual Fortran Reference Guide.

To see a solution's properties:

1. Select the solution in the Solution Explorer.
2. Right-click to bring up a context menu.
3. Select the *Properties* option.

   This action brings up the Property Pages dialog.

To see the properties for a project or file:

1. Select a project or a file in the Solution Explorer.
2. Right-click to bring up a context menu.
3. Select the *Properties* option.

   This action brings up the Property Pages dialog.

At the top of the Property Pages dialog there is a box labeled *Configuration*. In a PVF project, two configurations are created by default:

- The **Debug** configuration has properties set to build a version of your application that can be easily examined and controlled using the PVF debugger.
- The **Release** configuration has properties set so a version of your application is built with some general optimizations.

When a project is initially created, the Debug configuration is the active configuration. When you built the HelloWorld solution in Creating a Hello World Project, you built and ran the Debug configuration of your project. Let's look now at how to debug this application.

6. Run the Application Using the Debugger

To debug an application in PVF:

1. Set a breakpoint on the print statement in `ConsoleApp.f90`.

   To set a breakpoint, left-click in the far left side of the editor on the line where you want the breakpoint. A red circle appears to indicate that the breakpoint is set.
2. Select *Debug* | *Start Debugging* from the main menu to start the PGI Visual Fortran debug engine.

   The debug engine stops execution at the breakpoint set in Step 1.
3. Select *Debug* | *Step Over* to step over the print statement. Notice that the program output appears in a PGI Visual Fortran console window.
4. Select *Debug* | *Continue* to continue execution.

The program should exit.

For more information about building and debugging your application, refer to Build with PVF and Debug with PVF. Now that you have seen a complete example, let's take a look at more of the functionality available in several areas of PVF.

1.4. Using PVF Help

The PGI Visual Fortran User's Guide, PGI Visual Fortran Reference Manual, and PGI Fortran Reference are accessible online at PGI Documentation, www.pgroup.com/resources/docs/18.5/x86/index.htm.

Context-sensitive (<F1>) help is not currently supported in PVF.

1.5. PVF Sample Projects

The PVF installation includes several sample solutions, available from the PVF installation directory, typically in a directory called `Samples`:

```
$(VSInstallDir)\PGI Visual Fortran\Samples\
```

These samples provide simple demonstrations of specific PVF project and solution types.

In the `dlls` subdirectory of the `Samples` directory, you find this sample program:

pvf_dll
: Creates a DLL that exports routines written in Fortran.

In the `gpu` subdirectory of the `Samples` directory, you find these sample programs which require a PGI Accelerator License to compile and a GPU to run.

AccelPM_Matmul
: Uses directives from the PGI Accelerator Programming Model to offload a matmul computation to a GPU.

CUDAFor_Matmul
: Uses CUDA Fortran to offload a matmul computation to a GPU.

In the `interlanguage` subdirectory of the `Samples` directory, you find this sample program which requires that Visual C# be installed to build and run:

csharp_calling_pvfdll
: Calls a routine in a PVF DLL from a Visual C# test program.

In the `interlanguage` subdirectory of the `Samples` directory, you find these sample programs which require that Visual C++ be installed to build and run:

pvf_calling_vc
: Creates a solution containing a Visual C++ static library, where the source is compiled as C, and a PVF main program that calls it.

vcmain_calling_pvfdll
: Calls a routine in a PVF DLL from a main program compiled by VC++.

1.6. Compatibility

PGI Visual Fortran provides features that are compatible with those supported by older Windows Fortran products, such as Compaq® Visual Fortran. These include:

- Win32 API Support (`dfwin`)
- Unix/Linux Portability Support (`dflib, dfport`)
- Graphical User Interface Support

PVF provides access to a number of libraries that export C interfaces by using Fortran modules. This is the mechanism used by PVF to support the Win32 Application Programming Interface (API) and Unix/Linux portability libraries. If `C:` is your system drive, and <target> is your target system, such as win64, then source code containing the interfaces in these modules is located here:

```
C:\Program Files\PGI\<target>\<release_number>\src\
```

For more information about the specific functions in `dfwin`, `dflib`, and `dfport`, refer to the *Fortran Module / Library Interfaces for Windows* section in the PGI Visual Fortran Reference Manual.

1.6.1. Win32 API Support (dfwin)

The Microsoft Windows operating system interface (the system call and library interface) is known collectively as the Win32 API. This is true for both the 32-bit and 64-bit versions of Windows; there is no "Win64 API" for 64-bit Windows.

PGI Visual Fortran provides access to the Win32 API using Fortran modules. For details on specific Win32 API routines, refer to the Microsoft MSDN website.

For ease of use, the only module you need to use to access the Fortran interfaces to the Win32 API is `dfwin`. To use this module, simply add the following line to your Fortran code.

```
use dfwin
```

Table 2 lists all of the Win32 API modules and the Win32 libraries to which they correspond.

Table 2 PVF Win32 API Module Mappings

PVF Fortran Module	C Win32 API Lib	C Header File
advapi32	advapi32.lib	WinBase.h
comdlg32	comdlg32.lib	ComDlg.h
gdi32	gdi32.lib	WinGDI.h
kernel32	kernel32.lib	WinBase.h
shell32	shell32.lib	ShellAPI.h
user32	user32.lib	WinUser.h
winver	winver.lib	WinVer.h

PVF Fortran Module	C Win32 API Lib	C Header File
wsock32	wsock32.lib	WinSock.h

1.6.2. Unix/Linux Portability Interfaces (dflib, dfport)

PVF also includes Fortran module interfaces to libraries supporting some standard C library and Unix/Linux system call functionality. These functions are provided by the `dflib` and `dfport` modules. To utilize these modules add the appropriate `use` statement:

```
use dflib
```

```
use dfport
```

For more information about the specific functions in `dflib` and `dfport`, refer to 'Fortran Module/Library Interfaces for Windows' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

1.6.3. Windows Applications and Graphical User Interfaces

Programs that manage graphical user interface components using Fortran code are referred to as Windows Applications within PVF.

PVF Windows Applications are characterized by the lack of a `PROGRAM` statement. Instead, Windows Applications must provide a `WinMain` function like the following:

PVF WinMain for x64

```
integer(4) function WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
integer(8) hInstance
integer(8) hPrevInstance
integer(8) lpszCmdLine
integer(4) nCmdShow
```

`nCmdShow` is an integer specifying how the window is to be shown. For more details you can look up `WinMain` using the Microsoft MSDN website.

You can create a PVF Windows Application template by selecting `Windows Application` in the PVF New Project dialog. The project type of this name provides a default implementation of `WinMain`, and the project's properties are configured appropriately. You can also change the Configuration Type property of another project type to Windows Application using the General property page, described in the 'General Property Page' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf. If you do this, the configuration settings change to expect `WinMain` instead of `PROGRAM`, but a `WinMain` implementation is not provided.

Building Windows Applications from the Command Line

Windows Applications can also be built using a command line version of pgfortran. To enable this feature, add the `-winapp` option to the compiler driver command line when linking the application. This option causes the linker to include the correct libraries and object files needed to support a Windows Application. However, it does not add any additional system libraries to the link line. Add any required system libraries by adding the option `-defaultlib:<library name>` to the link command line for each library. For this option, <library name> can be any of the following: advapi32, comdlg32, gdi32, kernel32, shell32, user32, winver, or wsock32.

For more information about the specific functions in each of these libraries, refer to 'Fortran Module/Library Interfaces for Windows' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

Chapter 2. BUILD WITH PVF

This section describes how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment) to create and build a PVF project.

For information on general use of Visual Studio, see Miocrosoft's MSDN website. PVF runs within Visual Studio, so to invoke PVF you must invoke Visual Studio. If other languages such as Visual C++ or Visual Basic are installed, they will be available in the same instance of Visual Studio as PVF.

2.1. Creating a PVF Project

2.1.1. PVF Project Types

Once Visual Studio is running, you can use it to create a PGI Visual Fortran project. PVF supports a variety of project types:

- **Console Application** - An application (`.exe`) that runs in a console window, using text input and output.
- **Dynamic Library** - A dynamically-linked library file (`.dll`) that provides routines that can be loaded on-demand when called by the program that needs them.
- **Static Library** - An archive file (`.lib`) containing one or more object files that can be linked to create an executable.
- **Windows Application** - An application (`.exe`) that supports a graphical user interface that makes use of components like windows, dialog boxes, menus, and so on. The name of the program entry point for such applications is `WinMain`.
- **Empty Project** - A skeletal project intended to allow migration of existing applications to PVF. This project type does not include any source files. By default, an empty project is set to build an application (`.exe`).

2.1.2. Creating a New Project

To create a new project, follow these steps:

1. Select *File* | *New* | *Project* from the File menu.

The *New Project* dialog appears.

2. In the left-hand pane of the dialog, select *PGI Visual Fortran*.

   The right-hand pane displays the icons that correspond to the project types listed in Table 5.
3. Select the project type icon corresponding to the project type you want to create.
4. Name the project in the edit box labeled *Name*.

   > **Tip** The name of the first project in a solution is also used as the name of the solution itself.
5. Select where to create the project in the edit box labeled *Location*.
6. Click OK and the project is created.

Now look in the Solution Explorer to see the newly created project files and folders.

2.2. PVF Solution Explorer

PVF uses the standard Visual Studio Solution Explorer to organize files in PVF projects.

> **Tip** If the Solution Explorer is not already visible in the VS IDE, open it by selecting *View | Solution Explorer*.

Visual Studio uses the term project to refer to a set of files, build rules, and so on that are used to create an output like an executable, DLL, or static library. Projects are collected into a solution, which is composed of one or more projects that are usually related in some way.

PVF projects are reference-based projects, which means that although there can be folders in the representation of the project in the Solution Explorer, there are not necessarily any corresponding folders in the file system. Similarly, files added to the project can be located anywhere in the file system; adding them to the project does not copy them or move them to a project folder in the file system. The PVF project system keeps an internal record of the location of all the files added to a project.

2.3. Adding Files to a PVF Project

This section describes how to add a new file to a project and how to add an existing file to a project.

2.3.1. Add a New File

To add a new file to a PVF project, follow these steps:

1. Use the Solution Explorer to select the PVF project to which you want to add the new file.
2. Right-click on this PVF project to bring up a context menu.

3. Select *Add* => *New Item...*
4. In the *Add New Item* dialog box, select a file type from the available templates.
5. A default name for this new file will be in the *Name* box. Type in a new name if you do not want to use the default.
6. Click Add.

2.3.2. Add an Existing File

To add an existing file to a PVF project, follow these steps:

1. Use the Solution Explorer to select the PVF project to which you want to add the new file.
2. Right-click on this PVF project to bring up a context menu.
3. Select *Add* => *Existing Item...*
4. In the Browse window that appears, navigate to the location of the file you want to add.
5. Select the file and click Add.

> **Tip** You can add more than one file at a time by selecting multiple files.

2.4. Adding a New Project to a Solution

Each project produces one output, where an output is an executable, a static library, or a dynamic-link library (DLL). For example, if you want one solution to build both a library and an application that links against that library, you need two projects in the solution.

To add a project to a solution, follow these steps:

1. Use the Solution Explorer to select the solution.
2. Right-click on the solution to bring up a context menu.
3. Select *Add* => *New Project...*

   The Add New Project dialog appears. To learn how to use this dialog, refer to Creating a New Project.
4. In the *Add New Project* dialog box, select a project type from the available templates.
5. When you have selected and named the new project, click OK.

> Each project is specific to a single programming language, like PGI Visual Fortran or Microsoft Visual C++, but you can have projects that use different languages in the same solution.

2.5. Project Dependencies and Build Order

If your solution contains more than one project, set up the dependencies for each project to ensure that projects are built in the correct order.

To set project dependencies:

1. Right-click a project in the Solution Explorer.
2. From the resulting context menu select *Build Dependencies* (in older version of VS, *Project Dependencies* is not under *Build Dependencies*.

   The dialog box that opens has two tabs: Dependencies and Build Order.

   a. Use the Dependencies tab to put a check next to the projects on which the current project depends.
   b. Use the Build Order tab to verify the order in which projects will be built.

2.6. Configurations

Visual Studio projects are generally created with two default configurations: Debug and Release. The Debug configuration is set up to build a version of your application that can be easily debugged. The Release configuration is set up to build a generally-optimized version of your application. Other configurations may be created as desired using the Configuration Manager.

2.7. Platforms

In Visual Studio, the platform refers to the operating system for which you are building your application.

When you create a new project, you select its default platform. When more than one platform is available, you can add additional platforms to your project once it exists. To do this, you use the Configuration Manager.

2.8. Setting Global User Options

Global user options are settings that affect all Visual Studio sessions for a particular user, regardless of which project they have open. PVF supports several global user settings which affect the directories that are searched for executables, include files, and library files. To access these:

1. From the main menu, select *Tools | Options...*
2. From the Options dialog, expand *Projects and Solutions*.
3. Select *PVF Directories* in the dialog's navigation pane.

The PVF Directories page has two combo boxes at the top:

- **Platform** allows selection of the platform (i.e., x64).
- **Show directories for** allows selection of the search path to edit.

  Search paths that can be edited include the *Executable files* path, the *Include and module files* path, and the *Library files* path.

> **Tip** It is good practice to ensure that all three paths contain directories from the same release of the PGI compilers; mixing and matching different releases of the compiler executables, include files, and libraries can have undefined results.

2.9. Setting Configuration Options using Property Pages

Visual Studio makes extensive use of property pages to specify configuration options. Property pages are used to set options for compilation, optimization and linking, as well as how and where other tools like the debugger operate in the Visual Studio environment. Some property pages apply to the whole project, while others apply to a single file and can override the project-wide properties.

You can invoke the *Property Page* dialog in several ways:

- Select *Project | Properties* to invoke the property pages for the currently selected item in the Solution Explorer. This item may be a project, a file, a folder, or the solution itself.
- Right-click a project node in the Solution Explorer and select *Properties* from the resulting context menu to invoke that project's property pages.
- Right-click a file node in the Solution Explorer and select *Properties* from the context menu to invoke that file's property pages.

The Property Page dialog has two combo boxes at the top: **Configuration** and **Platform**. You can change the configuration box to *All Configurations* so the property is changed for all configurations.

> **Tip** A common error is to change a property like 'Additional Include Directories' for the Debug configuration but not the Release configuration, thereby breaking the build of the Release configuration.

In the PGI Visual Fortran Reference Manual, the 'Command-Line Options Reference' section contains descriptions of compiler options in terms of the corresponding command-line switches. For compiler options that can be set using the PVF property pages, the description of the option includes instructions on how to do so.

2.10. Property Pages

Properties, or configuration options, are grouped into property pages. Further, property pages are grouped into categories. Depending on the type of project, the set of available categories and property pages vary. The property pages in a PVF project are organized into the following categories:

- General
- Debugging
- Fortran
- Linker
- Librarian
- Resources
- Build Events
- Custom Build Step

Tip The Fortran, Linker and Librarian categories contain a Command Line property page where the command line derived from the properties can be seen. Options that are not supported by the PVF property pages can be added to the command line from this property page by entering them in the Additional Options field.

Table 3 shows the properties associated with each property page, listing them in the order in which you see them in the Properties dialog box. For a complete description of each property, refer to the *PVF Properties* section of the PGI Visual Fortran Reference Guide.

Table 3 Property Summary by Property Page

This Property Page...	Contains these properties...
General Property Page	Output Directory Intermediate Directory Extensions to Delete on Clean Configuration Type Build Log File Build Log Level
Debugging	Application Command Application Arguments Environment Merge Environment Accelerator Profiling MPI Debugging Working Directory [Serial] Number of Processes [Local MPI] Working Directory [Local MPI] Additional Arguments: mpiexec [Local MPI] Location of mpiexec [Local MPI]
Fortran | General	Display Startup Banner Additional Include Directories Module Path Object File Name Debug Information Format Optimization
Fortran | Optimization	Optimization Global Optimizations Vectorization

This Property Page...	Contains these properties...
	Inlining Use Frame Pointer Loop Unroll Count Auto-Parellelization
Fortran | Preprocessing	Preprocess Source File Additional Include Directories Ignore Standard Include Path Preprocessor Definitions Undefine Preprocessor Definitions
Fortran | Code Generation	Runtime Library
Fortran | Language	Fortran Dialect Treat Backslash as Character Extend Line Length Enable OpenMP Directives Enable OpenACC Directives OpenACC Autoparallelization OpenACC Required OpenACC Routineseq OpenACC Wait OpenACC Conformance Level OpenACC Sync MPI Enable CUDA Fortran CUDA Fortran Register Limit CUDA Fortran Use Fused Multiply-Adds CUDA Fortran Use Fast Math Library CUDA Fortran Debug CUDA Fortran Line Information CUDA Fortran Use LLVM Backend CUDA Fortran Unroll CUDA Fortran Flush to Zero CUDA Fortran Toolkit CUDA Fortran Compute Capability CUDA Fortran CC Fermi CUDA Fortran CC Kepler CUDA Fortran CC Maxwell CUDA Fortran CC Pascal CUDA Fortran CC Volta CUDA Fortran Keep Binary CUDA Fortran Keep Kernel Source CUDA Fortran Keep PTX CUDA Fortran PTXAS Info CUDA Fortran Generate RDC CUDA Fortran Emulation CUDA Fortran Madconst
Fortran | Floating Point Options	Floating Point Exception Handling

This Property Page...	Contains these properties...
	Floating Point Consistency Flush Denormalized Results to Zero Treat Denormalized Values as Zero IEEE Arithmetic
Fortran | External Procedures	Calling Convention String Length Arguments Case of External Names
Fortran | Libraries	Use MKL
Fortran | Target Processors	AMD Athlon AMD Barcelona AMD Bulldozer AMD Istanbul AMD Piledriver AMD Shanghai Intel Core 2 Intel Core 17 Intel Penryn Intel Pentium 4 Intel Sandy Bridge Generic x86-64 [x64 only]
Fortran | Target Accelerators	Target NVIDIA Tesla Tesla Register Limit Tesla Use Fused Multiple-Adds Tesla Use Fused Math Library Tesla LLVM Tesla Noattach Tesla Pin Host Memory Tesla Autocollapse Tesla Debug Tesla Lineinfo Tesla Unroll Tesla Required Tesla Flush to Zero Tesla Generate RDC Tesla CUDA Toolkit Tesla Compute Capability Tesla CC Fermi Tesla CC Kepler Tesla CC Maxwell Tesla CC Pascal Tesla CC Volta Tesla Keep Kernel Files Target Host
Fortran | Diagnostics	Warning Level Generate Assembly

This Property Page...	Contains these properties...
	Annotate Assembly Accelerator Information CCFF Information Fortran Language Information Inlining Information IPA Information Loop Intensity Information Loop Optimization Information LRE Information OpenMP Information Optimization Information Parallelization Information Unified Binary Information Vectorization Information
Fortran | Profiling	Function-Level Profiling Line-Level Profiling MPI Suppress CCFF Information Enable Limited Dwarf
Fortran | Runtime	Check Array Bounds Check Pointers Check Stack
Fortran | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Linker | General	Output File Additional Library Directories Stack Reserve Size Stack Commit Size Export Symbols
Linker | Input	Additional Dependencies
Linker | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Librarian | General	Output File Additional Library Directories Additional Dependencies
Librarian | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Resources | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Build Events | Pre-Build Event	Command Line

This Property Page...	Contains these properties...
	Description Excluded from Build
Build Events | Pre-Link Event	Command Line Description Excluded from Build
Build Events | Post-Build Event	Command Line Description Excluded from Build
Custom Build Step | General	Command Line Description Outputs Additional Dependencies

2.11. Setting File Properties Using the Properties Window

Properties accessed from the Property Pages dialog allow you to change the configuration options for a project or file. The term property, however, has another meaning in the context of the Properties Window. In the Properties Window *property* means attribute or characteristic.

To see a file's properties, do this:

1. Select the file in the Solution Explorer.
2. From the *View* menu, open the *Properties Window*.

Some file properties can be modified, while others are read-only.

The values of the properties in the Properties Window remain constant regardless of the Configuration (Debug, Release) or Platform (x64) selected.

Table 4 lists the file properties that are available in a PVF project.

Table 4 PVF Project File Properties

This property...	Does this...
Name	Shows the name of the selected file.
Filename	Shows the name of the selected file.
FilePath	Shows the absolute path to the file on disk. (Read-only)
FileType	Shows the registered type of the file, which is determined by the file’s extension. (Read-only)
IsCUDA	Indicates whether the file is considered a CUDA Fortran file.

This property...	Does this...
	`True` indicatesthe file's extension is `.cuf` or the Enable CUDA Fortran property is set to Yes (Read-only). `False` indicatesthe file is not a CUDA Fortran file.
IsFixedFormat	Determines whether the Fortran file is fixed format. `True` indicates fixed format and `False` indicates free format. To change whether a source file is compiledas fixed or free format source, set this property appropriately. PVF initially uses file extensions to determine format style: the `.f` and `.for` extensions imply fixed format, while other extensions such as `.f90` or `.f95` imply free format. The 'C' and '*' comment charactersare only valid for fixed format compilation.
IsIncludeFile	A boolean value that indicates if the file is an include file. When `True`,PVF considers the file to be an include file and it does not attempt to compile it. When `False`, if the filenamehas a supported Fortran or Resource file extension, PVF compiles the file as part of the build. **Tip** You can use thisproperty to exclude a source file from a build.
IsOutput	Indicates whether a file is produced by the build. (Read-only)
ModifiedDate	Contains the date and time that the file was last saved to disk. (Read-only)
ReadOnly	Indicates the status of the Read-Only attribute of the file on disk.
Size	Describes the size of the file on disk.

2.12. Setting Fixed Format

Some Fortran source is written in fixed-format style. If your fixed-format code does not compile, check that it is designated as fixed-format in PVF.

To check fixed-format in PVF, follow these steps:

1. Use the Solution Explorer to select a file: View | Solution Explorer.
2. Open the Properties Window: View | Properties Window.
3. From the dropdown list for the file property *IsFixedFormat*, select *True*.

2.13. Building a Project with PVF

Once a PVF project has been created, populated with source files, and any necessary configuration settings have been made, the project can be built. The easiest way to start a build is to use the *Build | Build Solution* menu selection; all projects in the solution will be built.

If there are compile-time errors, the *Error List* window is displayed, showing a summary of the errors that were encountered. If the error message shows a line number, then double-clicking the error record in the *Error List* window will navigate to the location of the error in the editor.

When a project is built for the first time, PVF must determine the build dependencies. Build dependencies are the result of `USE` or `INCLUDE` statements or `#include` preprocessor directives in the source. In particular, if file A contains a `USE` statement referring to a Fortran module defined in file B, file B must be compiled successfully before file A will compile.

To determine the build dependencies, PVF begins compiling files in alphabetical order. If a compile fails due to an unsatisfied module dependency, the offending file is placed back on the build queue and a message is printed to the *Output Window*, but not to the *Error List*. In a correct Fortran program, all dependencies will eventually be met, and the project will be built successfully. Otherwise, errors will be printed to the *Error List* as usual.

Unless the build dependencies change, subsequent builds use the build dependency information generated during the course of the initial build.

2.13.1. Order of PVF Build Operations

In the default PVF project build, the build operations are executed in the following order:

1. Pre-Build Event
2. Custom Build Steps for Files
3. Build Resources
4. Compile Fortran Files to Objects (using the PGI Fortran compiler)
5. Pre-Link Event
6. Build Output Files (using linker or librarian)
7. Custom Build Step for Project
8. Post-Link Event

2.14. Build Events and Custom Build Steps

PVF provides default build rules for Fortran files and Resource files. Other files are ignored unless a build action is specified using a Build Event or a Custom Build Step.

2.14.1. Build Events

Build events allow definition of a specific command to be executed at a predetermined point during the project build. You define build events using the property pages for the project. Build events can be specified as Pre-Build, Pre-Link, or Post-Build. For specific information about where build events are run in the PVF build, refer to Order of PVF

Build Operations. Build events are always run unless the project is up to date. There is no dependency checking for build events.

2.14.2. Custom Build Steps

Custom build steps are defined using the 'Custom Build Step Property.' You can specify a custom build step for an entire project or for an individual file, provided the file is not a Fortran or Resource file.

When a custom build step is defined for a project, dependencies are not checked during a build. As a result, the custom build step only runs when the project itself is out of date. Under these conditions, the custom build step is very similar to the post-build event.

When a custom build step is defined for an individual file, dependencies may be specified. In this case, the dependencies must be out of date for the custom build step to run.

> The 'Outputs' property for a file-level custom build step must be defined or the custom build step is skipped.

2.15. PVF Build Macros

PVF implements a subset of the build macros supported by Visual C++ along with a few PVF-specific macros. The macro names are not case-sensitive, and they should be usable in any string field in a property page. Unless otherwise noted, macros that evaluate to directory names end with a trailing backslash ('\').

In general these items can only be changed if there is an associated PVF project or file property. For example, $(VCInstallDir) cannot be changed, while $(IntDir) can be changed by modifying the General | Intermediate Directory property.

For the names and descriptions of the build macros that PVF supports, refer to the 'PVF Build Macros' section in the PGI Visual Fortran Reference Manual.

2.16. Static and Dynamic Linking

PVF supports both static and dynamic linking to the PGI and Microsoft runtime.

The *Fortran | Code Generation | Runtime Library* property in a project's property pages determines which runtime library the project targets.

- For executable and static library projects, the default value of this property is **static linking** (-Bstatic). A statically-linked executable can be run on any system for which it is built; neither the PGI nor the Microsoft redistributable libraries need be installed on the target system.
- For dynamically linked library projects, the default value of this property is **dynamic linking** (-Bdynamic). A dynamically-linked executable can only be run on a system on which the PGI and Microsoft runtime redistributables have been installed.

For more information on deploying PGI-compiled applications to other systems, refer to Distributing Files – Deployment.

2.17. VC# Interoperability

If Visual C# is installed along with PVF, Visual Studio solutions containing both PVF and VC# projects can be created. Each project, though, must be purely PVF or VC#; Fortran and C# code cannot be mixed in a single project.

For an example of how to create a Fortran and VC# solution, refer to the PVF sample project `csharp_calling_pvfdll`.

> Because calling Visual C++ code (as opposed to C code) from Fortran is very complicated, it is only recommended for the advanced programmer. Further, to make interfaces easy to call from Fortran, Visual C++ code should export the interfaces using `extern "C"`.

2.18. VC++ Interoperability

If Visual C++ is installed along with PVF, Visual Studio solutions containing both PVF and VC++ projects can be created. Each project, though, must be purely PVF or VC++; Fortran and C/C++ code cannot be mixed in a single project. This constraint is purely an organizational issue. Fortran subprograms may call C functions and C functions may call Fortran subprograms as outlined in Inter-language Calling.

For an example of how to create a solution containing a VC++ static library, where the source is compiled as C, and a PVF main program that calls into it, refer to the PVF sample project `pvf_calling_vc`.

> Because the process of calling Visual C++ code (as opposed to C code) from Fortran is very complicated, it is only recommended for the advanced programmer. Further, to make interfaces easy to call from Fortran, Visual C++ code should export the interfaces using `extern "C"`.

2.19. Linking PVF and VC++ Projects

If you have multiple projects in a solution, be certain to use the same type of runtime library for all the projects. Further, if you have Microsoft VC++ projects in your solution, you need to be certain to match the runtime library types in the PVF projects to those of the VC++ projects.

PVF's property *Fortran | Code Generation | Runtime Library* corresponds to the Microsoft VC++ property named *C/C++ | Code Generation | Runtime Library*. Table 5 lists the appropriate combinations of Runtime Library property values when mixing PVF and VC++ projects.

Table 5 Runtime Library Values for PVF and VC++ Projects

If PVF uses ...	VC++ should use...
Multi-threaded (-Bstatic)	Multi-threaded (/MT)
Multi-threaded DLL (-Bdynamic)	Multi-threaded DLL (/MD)
Multi-threaded DLL (-Bdynamic)	Multi-threaded debug DLL (/MDd)

2.20. Common Link-time Errors

The runtime libraries specified for all projects in a solution should be the same. If both PVF and VC++ projects exist in the same solution, the runtime libraries targeted should be compatible.

Keep in mind the following guidelines:

- Projects that produce DLLs should use the Multi-threaded DLL (-Bdynamic) runtime.
- Projects that produce executables or static libraries can use either type of linking.

The following examples provide a look at some of the link-time errors you might see when the runtime library targeted by a PVF project is not compatible with the runtime library targeted by a VC++ project. To resolve these errors, refer to Table 5 and set the Runtime Library properties for the PVF and VC++ projects accordingly.

Errors seen when linking a PVF project using -Bstatic and a VC++ library project using /MDd:

```
MSVCRTD.lib(MSVCR80D.dll) : error LNK2005: _printf already defined in
 libcmt.lib(printf.obj)  LINK : warning LNK4098: defaultlib 'MSVCRTD'
 conflicts with use of other libs; use /NODEFAULTLIB:library test.exe : fatal
 error LNK1169: one or more multiply defined symbols found
```

Errors seen when linking a PVF project using -Bstatic and a VC++ project using /MTd:

```
LIBCMTD.lib(dbgheap.obj) : error LNK2005: _malloc already defined in
 libcmt.lib(malloc.obj) ... LINK : warning LNK4098: defaultlib 'LIBCMTD'
 conflicts with use of other libs; use /NODEFAULTLIB:library test.exe : fatal
 error LNK1169: one or more multiply defined
```

2.21. Migrating an Existing Application to PVF

An existing non-PVF Fortran application or library project can be migrated to PVF. This section provides a rough outline of how one might go about such a migration.

> **Tip** Depending on your level of experience with Visual Studio and the complexity of your existing application, you might want to experiment with a practice project first to become familiar with the project directory structure and the process of adding existing files.

Start your project migration by creating a new Empty Project. Add the existing source and include files associated with your application to the project. If some of your source files build a library, while other files build the application itself, you will need to create a separate project within your solution for the files that build the library.

Set the configuration options using the property pages. You may need to add include paths, module paths, library dependency paths and library dependency files. If your solution contains more than one project, you will want to set up the dependencies between projects to ensure that the projects are built in the correct order.

When you are ready to try a build, select *Build | Build Solution* from the main menu. This action starts a full build. If there are compiler or linker errors, you will probably have a bit more build or configuration work to do.

2.22. Fortran Editing Features

PVF provides several Fortran-aware features to ease the task of entering and examining Fortran code in the Visual Studio Editor.

Source Colorization—Fortran source is colorized, so keywords, comments, and strings are distinguished from other language elements. You can use the *Tools | Options | Environment | Fonts and Colors* dialog to assign colors for identifiers and numeric constants, and to modify the default colors for strings, keywords and comments.

Method Tips—Fortran intrinsic functions are supported with method tips. When an opening parenthesis is entered in the source editor following an intrinsic name, a method tip pop-up is displayed that shows the data types of the arguments to the intrinsic function. If the intrinsic is a generic function supporting more than one set of arguments, the method tip window supports scrolling through the supported argument lists.

Keyword Completion—Fortran keywords are supported with keyword completion. When entering a keyword into the source editor, typing `<CTRL>+<SPACE>` will open a pop-up list displaying the possible completions for the portion of the keyword entered so far. Use the up or down arrow keys or the mouse to select one of the displayed items; type `<ENTER>` or double-click to enter the remainder of the highlighted keyword into the source. Type additional characters to narrow the keyword list or use `<BACKSPACE>` to expand it.

Chapter 3. DEBUG WITH PVF

PVF utilizes the Visual Studio debugger for debugging Fortran programs. PGI has implemented a custom debug engine that provides the language-specific debugging capability required for Fortran. This debug engine also supports Visual C++.

The Debug configuration is usually used for debugging. By default, this configuration will build the application so that debugging information is provided.

The debugger can be started by selecting *Debug* | *Start Debugging*. Then use the Visual Studio debugger controls as usual.

3.1. Windows Used in Debugging

Visual Studio uses a number of different windows to provide debugging information. Only a subset of these is opened by default in your initial debugging session. Use the *Debug* | *Windows* menu option to see a list of all the windows available and to select the one you want to open.

This section provides an overview of most of the debugging windows you can use to get information about your debug session, along with a few tips about working with some of these windows.

3.1.1. Autos Window

The autos window provides information about a changing set of variables as determined by the current debugging location. This window is supported for VC++ code but will not contain any information when debugging in a Fortran source file.

3.1.2. Breakpoints Window

The breakpoints window contains all the breakpoints that have been set in the current application. You use the breakpoints window to manage the application's breakpoints.

This window is available even when the application is not being debugged.

You can disable, enable or delete any or all breakpoints from within this window.

- Double-clicking on a breakpoint opens the editor to the place in the source where the breakpoint is set.
- Right-clicking on a breakpoint brings up a context menu display that shows the conditions that are set for the breakpoint. You can update these conditions via this display.
- During debugging, each breakpoint's status is shown in this window.

Breakpoint States

A breakpoint can be enabled, disabled, or in an error state. A breakpoint in an error state indicates that it failed to bind to a code location when the program was loaded. An error breakpoint can be caused by a variety of things. Two of the most common reasons a breakpoint fails to bind are these:

- The code containing the breakpoint may be in a module (DLL) that has not yet been loaded.
- A breakpoint audience may contain a syntax error.

Breakpoints in Multi-Process Programs

When debugging a multi-process program, each user-specified breakpoint is bound on a per-process basis. When this situation occurs, the breakpoints in the breakpoints window can be expanded to reveal each bound breakpoint.

3.1.3. Call Stack Window

The call stack window shows the call stack based on the current debugging location. Call frames are listed from the top down, with the innermost function on the top. Double-click on a call frame to select it.

- The yellow arrow is the *instruction pointer*, which indicates the current location.
- A green arrow beside a frame indicates the frame is selected but is not the current frame.

3.1.4. Disassembly Window

The disassembly window shows the assembly code corresponding to the source code under debug.

Using *Step* and *Step Into* in the disassembly window moves the instruction pointer one assembly instruction instead of one source line. Whenever possible, source lines are interleaved with disassembly.

3.1.5. Immediate Window

The immediate window provides direct communication with the debug engine. You can type `help` in this window to get a list of supported commands.

Variable Values in Multi-Process Programs

When debugging a multi-process program, use the `print` command in the immediate window with a process/thread set to display the values of a variable across all processes at once. For example, the following command prints the value of `iVar` for all processes and their threads.

```
[*.*] print iVar
```

3.1.6. Locals Window

The locals window lists all variables in the current scope, providing the variable's name, value, and type. You can expand variables of type array, record, structure, union and derived type variables to view all members. The variables listed include any Fortran module variables that are referenced in the current scope.

3.1.7. Memory Window

The memory window lists the contents of memory at a specified address. Type an address in memory into the memory window's Address box to display the contents of memory at that address.

3.1.8. Modules Window

In Visual Studio, the term *module* means a program unit such as a DLL. It is unrelated to the Fortran concept of module.

The modules window displays the DLLs that were loaded when the application itself was loaded. You can view information such as whether or not symbol information is provided for a given module.

3.1.9. Output Window

The output window displays a variety of status messages. When building an application, this window displays build information. When debugging, the output window displays information about loading and unloading modules, and exiting processes and threads.

The output window does not receive application output such as standard out or standard error. In serial and local MPI debugging, such output is directed to a console window.

3.1.10. Processes Window

The processes window displays each process that is currently being debugged. For serial debugging, there is only one process displayed. For MPI debugging, the number of processes specified in the Debugging property page determines the number of processes that display in this window. The Title column of the processes window contains the rank of each process, as well as the name of the system on which the process is running and the process id.

Switching Processes in Multi-Process Programs

Many of the debugging windows display information for one process at a time. During multi-process debugging, the information in these windows pertains to the process with focus in the processes window. The process with focus has a yellow arrow next to it.

You can change the focus from one process to another by selecting the desired process in one of these ways:

- Double-click on the process.
- Highlight the process and press <Enter>.

3.1.11. Registers Window

The registers window is available during debugging so you can see the value of the OS registers. Registers are shown in functional groups. The first time you use the registers window, the CPU registers are shown by default.

- To show other register sets, follow these steps:
  1. Right-click in the registers window to bring up a context menu.
  2. From the context menu, select the group of registers to add to the registers window display.
- To remove a group from the display, follow these steps:
  1. Right-click in the registers window to bring up a context menu.
  2. From the context menu, deselect the group of registers to remove from the registers window display.

3.1.12. Threads Window

The threads window lists the active threads in the current process. Threads are named by process and thread rank using the form "process.thread".

> Not all threads may be executing in user code at any given time.

3.1.13. Watch Window

You use the watch window during debugging to display a user-selected set of variables.

> If a watched variable is no longer in scope, its value is no longer valid in the watch window, although the variable itself remains listed until you remove it.

3.2. Variable Rollover

Visual Studio provides a debugging feature called *variable rollover*. This feature is available when an application in debug mode stops at a breakpoint or is otherwise suspended. To activate variable rollover, use the mouse pointer to hover over a variable in the source code editor. After a moment, the value of the variable appears as a data tip next to the mouse pointer.

The first data tip that you see is often upper level information, such as an array address or possibly the members of a user-defined type. If additional information is available for a variable, you see a plus sign in the data tip. Hovering over the plus sign expands the information. Once the expansion reaches the maximum number of lines available for display, about fifteen lines, the data tip has up and down triangles which allow you to scroll to see additional information.

You can use variable rollover to obtain information about scalars, arrays, array elements, as well as user-defined type variables and their members.

3.2.1. Scalar Variables

If you roll over a scalar variable, such as an integer or a real, the data tip displays the scalar’s value.

3.2.2. Array Variables

If you roll over an array, the data tip displays the array’s address.

To see the elements of an array, either roll over the specific array element’s subscript operator (parenthesis), or roll over the array and then expand the data tip by moving the mouse over the plus sign in the data tip. The expanded view displays the individual array elements.

The data tip can display up to about fifteen array elements at a time. For arrays with more than fifteen elements, use the up and down arrows on the top and bottom of the expanded data tip to scroll through the other elements.

Fortran character arrays work slightly differently.

- When rolling over a single element character array, the data tip displays the value of the string. To see the individual character elements, expand the string.
- When rolling over a multi-element character array, the initial data tip contains the array’s address. To see the elements of the array, expand the array. Each expanded element appears as a string, which is also expandable.

3.2.3. User-Defined Type Variables

User-defined types include derived types, records, structs, and unions. When rolling over a user-defined type, the initial data tip displays a condensed form of the value of the user-defined type variable, which is also expandable.

To see a member of a user-defined type, you can either roll over the specific user-defined variable directly, or roll over the user-defined type and then expand the data tip by moving the mouse over the plus sign in the data tip. The expanded view displays the individual members of the variable and their values.

The data tip can display up to about fifteen user-defined type members at a time. For user-defined types with more than fifteen members, use the up and down arrows on the top and bottom of the expanded data tip to scroll through the other members.

3.3. Debugging an MPI Application in PVF

PVF has full debugging support for MPI applications running locally. For specific information on how to do this, refer to Debug an MPI Application.

3.4. Attaching the PVF Debugger to a Running Application

PGI Visual Fortran can debug a running application using the PVF "Attach to Process" option. PVF supports attaching to Fortran applications built for native Windows systems.

PVF includes PGI compilers that build on native Windows applications. A PVF installation is all that is required to use PVF to attach to PGI-compiled native Windows applications.

The following instructions describe how to use PVF to attach to a running native Windows application. As is often true, the richest debugging experience is obtained if the application being debugged has been compiled with debug information enabled.

3.4.1. Attach to a Native Windows Application

To attach to a native Windows application, follow these steps:

1. Open PVF from the Start menu, invoke PVF as described in PVF on the Start Screen and Start Menu.
2. From the main *Tools* menu, select *Attach to Process...*
3. In the *Attach to:* box of the *Attach to Process* dialog, verify that **PGI Debug Engine** is selected.

   If it is not selected, follow these steps to select it:

   1. Click Select.
   2. In the *Select Code* dialog box that appears, choose *Debug these code types*.
   3. Deselect any options that are selected and select *PGI Debug Engine*.
   4. Click OK.
4. Select the application to which you want to attach PVF from the *Available Processes* box in the *Attach to Process* dialog.

   This area of the dialog box contains the system's running processes. If the application to which you want to attach PVF is missing from this list, try this procedure to locate it:

1. Depending on where the process may be located, select *Show processes in all sessions* or *Show processes from all users*. You can select both.
2. Click Refresh.

5. With the application to attach to selected, click *Attach*.

PVF should now be attached to the application.

To debug, there are two ways to stop the application:

- Set a breakpoint using *Debug* | *New Breakpoint* | *Break at Function...* and let execution stop when the breakpoint is hit.

  Tip Be certain to set the breakpoint at a line in the function that has yet to be executed.

- Use *Debug* | *Break All* to stop execution.

  With this method, if you see a message box appear that reads `There is no source code available for the current location`, click OK. Use *Step Over (F10)* to advance to a line for which source is available.

To detach PVF from the application and stop debugging, select *Debug* | *Stop Debugging*.

3.5. Using PVF to Debug a Standalone Executable

You can invoke the PVF debug engine to debug an executable that was not created by a PVF project. To do this, you invoke Visual Studio from a command shell with special arguments implemented by PVF. You can use this method in any native Windows command prompt environment.

PGI Visual Fortran includes PGI compilers that build native Windows applications. A PVF installation is all that is required to use the PVF standalone executable debugging feature with PGI-compiled native Windows applications. The following instructions describe how to invoke the PGI Visual Fortran debug engine from a native Windows prompt.

Tip The richest debugging experience is obtained when the application being debugged has been compiled and linked with debug information enabled.

3.5.1. Launch PGI Visual Fortran from a Native Windows Command Prompt

To launch PGI Visual Fortran from a native Windows Command Prompt, follow these steps:

1. Set the environment by opening a PVF Command Prompt window using the PVF Start menu, as described in Shortcuts to Launch PVF.
    - To debug a 64-bit executable, choose the 64-bit command prompt: *PVF Cmd (64)*.

      The environment in the option you choose is automatically set to debug a native Windows application.
2. Start PGI Visual Fortran using the executable `devenv.exe`.

   If you followed Step 1 to open the PVF Command Prompt, this executable should already be on your path.

In the PVF Command Prompt window, you must supply the switch `/PVF:DebugExe`, your executable, and any arguments that your executable requires. The following examples illustrate this requirement.

Use PVF to Debug an Application

This example uses PVF to debug an application, `MyApp1.exe`, that requires no arguments.

```
CMD> devenv /PVF:DebugExe MyApp1
```

Use PVF to Debug an Application with Arguments

This example uses PVF to debug an application, `MyApp2.exe`, and pass it two arguments: arg1, arg2.

```
CMD> devenv /PVF:DebugExe MyApp2 arg1 arg2
```

Once PVF starts, you should see a Solution and Project with the same name as the name of the executable you passed in on the command line, such as `MyApp2` in the previous example.

You are now ready to use PGI Visual Fortran after a command line launch, as described in the next section.

3.5.2. Using PGI Visual Fortran After a Command Line Launch

Once you have started PVF from the command line, it does not matter how you started it, you are now ready to run and debug your application from within PVF.

To run your application from within PVF, from the main menu, select *Debug | Start Without Debugging*.

To debug your application using PVF:

1. Set a breakpoint using the *Debug | New Breakpoint | Break at Function* dialog box.
2. Enter either a function or a function and line that you know will be executed.

   > **Tip** You can always use the routine name `MAIN` for the program's entry point (i.e. main program) in a Fortran program compiled by PGI compilers.
3. Start the application using *Debug | Start Debugging*.

When the debugger hits the breakpoint, execution stops and, if available, the source file containing the breakpoint is opened in the PVF editor.

3.5.3. Tips on Launching PVF from the Command Line

If you choose to launch PVF from a command line, here are a few tips to help you be successful:

- The path to the executable you want to debug must be specified using a full or relative path. Further, paths containing spaces must be quoted using double quotes (").
- If you specify an executable that does not exist, PVF starts up with a warning message and no solution is created.
- If you specify a file to debug that exists but is not in an executable format, PVF starts up with a warning message and no solution is created.

Chapter 4. USING MPI IN PVF

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

PGI provides MPI support with PGI compilers and tools. You can build, run, debug, and profile MPI applications on Windows using PVF and Microsoft's implementation of MPI, MS-MPI. This section describes how to use these capabilities and indicates some of their limitations, provides the requirements for using MPI in PVF, explains how to compile and enable MPI execution, and describes how to launch, debug, and profile your MPI application. In addition, there are tips on how to get the most out of PVF's MPI capabilities.

4.1. MPI Overview

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. Further, distributed execution of a program does not necessarily mean that an MPI job must run on multiple machines.

PVF has built-in support for Microsoft's version of MPI: MS-MPI, on single systems. PVF does not support using MS-MPI on Windows clusters.

4.2. System and Software Requirements

To use PVF's MPI capabilities, MS-MPI must be installed on your system. The MS-MPI components include headers, libraries, and `mpiexec`, which PVF uses to launch MPI applications. The 2018 release of PVF includes a version of MS-MPI that is installed automatically when PVF is installed. MS-MPI can also be downloaded directly from Microsoft.

4.3. Compile using MS-MPI

The PVF Fortran | Language | MPI property enables MPI compilation and linking with the Microsoft MPI headers and libraries. Set this property to *Microsoft MPI* to enable an MPI build.

4.4. Enable MPI Execution

Once your MPI application is built, you can run and debug it. The PVF Debugging property page is the key to both running and debugging an MPI application. For simplicity, in this section we use the term *execution* to mean either running or debugging the application.

Use the MPI Debugging property to determine the type of execution you need, provided you have the appropriate system configuration and license.

4.4.1. MPI Debugging Property Options

The MPI Debugging property can be set to either of these options: Disabled or Local.

Disabled
When *Disabled* is selected, execution is performed serially.

Local
When *Local* is selected, MPI execution is performed locally. That is, multiple processes are used but all of them run on the local host.

Additional MPI properties become available when you select the Local MPI Debugging option. For more information about these properties, refer to the 'Debugging Property Page' in the PGI Visual Fortran Reference Manual.

4.5. Launch an MPI Application

As soon as you have built your MPI application, and selected Local MPI Debugging, you can launch your executable using the *Debug | Start Without Debugging* menu option.

PVF uses Microsoft's version of `mpiexec` to support Local MPI execution.

4.6. Debug an MPI Application

To debug your MPI application, select *Debug | Start Debugging* or hit F5. As with running your MPI application, PVF uses `mpiexec` for Local MPI jobs.

PVF's style of MPI debugging can be described as 'run altogether.' With this style of debugging, execution of all processes occurs at the same time. When you select *Continue*, all processes are continued. When one process hits a breakpoint, it stops. The other processes do not stop, however, until they hit a breakpoint or some other type of barrier.

When you select *Step*, all processes are stepped. Control returns to you as soon as one or more processes finish its step. If some process does not finish its step when the other processes are finished, it continues execution until it completes.

Chapter 5. GETTING STARTED WITH THE COMMAND LINE COMPILERS

This section describes how to use the command-line PGI compilers. The PGI Visual Fortran IDE invokes the PGI compilers when you build a PVF project. You can also invoke the compilers directly from a command prompt which you can launch from the Start menu, as described in Shortcuts to Launch PVF.

5.1. 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 x86-64 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.

5.2. 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 a filename based on the name of the first source or object file on the command line. However, you can specify an output file name by using the -o option.

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

   ```
   $ 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

5.3. Invoking the Command-level PGI Compilers

To translate and link a Fortran language program, the `pgf77`, `pgf95`, and `pgfortran` 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.

5.3.1. 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.

5.3.2. 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.

5.3.3. Fortran Directives

You can insert Fortran directives 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, refer to Using OpenMP and Using Directives.

5.4. 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.

5.4.1. Input Files

You can specify assembly-language files, preprocessed source files, Fortran 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.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.fpp`
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 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.s`
indicates an assembly-language file.

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

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

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

filename.dylib
(macOS systems only) indicates a dynamically-linked library.

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

Files with a `.fpp` suffix are first preprocessed by the Fortran compilers and the output is passed to the compilation phase. The Fortran preprocessor is built in to the Fortran compilers. This design ensures consistency in the preprocessing step regardless of the type or revision of operating system under which you're compiling.

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

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.

5.4.2. Output Files

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

If you use option -F (Fortran 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 6 Option Descriptions

Option	Stop After	Input	Output
-E	preprocessing	Source files	preprocessed file to standard out
-F	preprocessing	Source files	preprocessed file (`.f`)
-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 (`.obj`)
none	linking	Source files, or preprocessed files, assembly-language files, object files, or libraries	executable file (`.exe`)

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

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

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

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

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

5.5. Fortran Data Types

The PGI Fortran 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 x64 processor-based systems, refer to 'Fortran, C, and C++ Data Types' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

For more information on x86-64-specific data representation, refer to the *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.) listed in the 'Related Publications' section in the Preface.

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

5.6. 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 `pgf77`, `pgf95`, or `pgfortran`. Parallel programs of this variety can be run on shared-memory parallel (SMP) systems such as dual-core or multi-processor workstations.
- OpenMP shared-memory parallel programs compiled using the -mp option to `pgf77`, `pgf95`, or `pgfortran`. 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.

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.

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

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

5.7. 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.

5.7.1. 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.

5.7.2. 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 Windows, these files are named `mypgf77rc`, `mypgf90rc`, `mypgf95rc`, `mypgfortranrc` and `mypgccrc`.

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

5.8. 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 allow users to place hints in the source code to help the compiler generate better assembly code. You typically use directives 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 typically stays in effect from the point where included until the end of the compilation unit or until another directive changes its status. For more information on directives, refer to Using OpenMP and Using Directives.
- 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.

Chapter 6. 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

6.1. 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.

6.1.1. 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf contains this information.

6.1.2. 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
```

6.1.3. 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
```

6.2. 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

6.3. 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.

6.3.1. 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 for 64-bit targets. 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 7 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 8 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, even for FORTRAN 77 code, and the -fast option.

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

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

6.4. Targeting Multiple Systems—Using the -tp Option

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

Processor-specific optimizations can be specified or limited explicitly by using the `-tp` option. Thus, it is possible to create executables that are usable on previous generation systems. Using a `-tp` flag option of k8 or p7 produces an executable that runs on most x86-64 hardware in use today.

For more information about the `-tp` option, refer to the `-tp <target> [,target...]` description in the 'Command-Line Options Reference' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

6.5. 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.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 9 Commonly Used Command-Line Options

Use this option...	To do this...
`-fast` `-fastsse`	These options create a generally optimal set of flags for targets that support SIMD capability. They incorporate optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization with 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.
`-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.
`-Mpfi` or `-Mpfo`	Enable profile feedback driven optimizations
`-Mkeepasm`	Keeps the generated assembly files.

Use this option...	To do this...
`-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.
`-o`	Names the output file.
`-O <level>`	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
`-tp <target> [,target...]`	Specify the target processor(s); for the 64-bit compilers, more than one target is allowed, and enables generation of PGI Unified Binary executables.

Chapter 7. 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. In PVF, you use the Fortran | Optimization property page to specify optimization levels; on the command line, the options you commonly use include -O, -Mvect, and -Mconcur. In addition, you can use several of the -M<pgflag> switches to control specific types of optimization and parallelization. You can set the options not supported by the Fortran | Optimization property page by using the *Additional Options* field of the Fortran | Command Line property page. For more information, refer to *Fortran Property Pages* section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

> As of the PGI 16.3 release, -Mipa has been disabled on Windows.

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

This chapter describes these optimization options:

-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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

7.1. 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 x86-64 architecture, instruction set and registers.

For discussion purposes, we categorize optimization:

Local Optimization
Global Optimization
Loop Optimization
Interprocedural Analysis (IPA) and Optimization
Optimization Through Function Inlining
Profile Feedback Optimization (PFO)

7.1.1. 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.

7.1.2. 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.

7.1.3. 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 SSE instructions which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

7.1.4. Interprocedural Analysis (IPA) and Optimization

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

7.1.5. 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.

7.1.6. 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.

7.2. 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 x86-64 platforms. For more information on debugging, refer to the PGI Debugger User's Guide, www.pgroup.com/resources/docs/18.5/pdf/pgi18dbug.pdf.

> As of the PGI 16.3 release,
> ```
> -Mipa
> ```
> has been disabled on Windows.

In PVF, similar options may be accessed using the Optimization property in the Fortran | Optimization property page. For more information on these property pages, refer to the *Optimization* section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

By experimenting with individual compiler options on a file-by-file basis, further significant performance gains can sometimes be realized. However, depending on the coding style, individual optimizations can sometimes cause slowdowns, and must be used carefully to ensure performance improvements.

In PVF, you may access the -O3, -Minline, and -Mconcur options by using the Global Optimizations, Inlining, and Auto-Parallelization properties on the Fortran | Optimization property page, respectively. For more information on these property pages, refer to the *Optimization* section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

7.2.1. -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
```

In PVF these options may be accessed via the Fortran | Command Line property page, or perhaps more appropriately for the `-help` option via a Build Event or Custom Build Step. For more information on these property pages, refer to the 'Command Line' section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

7.2.2. -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, SSE instructions , vectorization, parallelization, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

For more information on `-Minfo`, refer to 'Optimization Controls' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

7.2.3. -Mneginfo

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

In PVF, you can use the Warning Level property available in the Fortran | General property page to specify the option `-Mneginfo`.

For more information on `-Mneginfo`, refer to 'Optimization Controls' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

7.2.4. -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.

7.2.5. -v

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

7.2.6. 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/18.5/pdf/pgi18profug.pdf.

7.3. 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.

7.4. Local and Global Optimization

This section describes local and global optimization.

7.4.1. -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
```

7.5. 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.

In PVF, this option is accessed using the Loop Unroll Count property in the Fortran | Optimization property page. For more information on these property pages, refer to 'Fortran Optimization' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

7.6. 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 *Streaming SIMD Extensions (SSE)* instructions on x86 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.

In PVF, you can access the basic forms of this option using the Vectorization property in the Fortran | Optimization property page. For more advanced use of this option, use the Fortran | Command Line property page. For more information on these property pages, refer to *Fortran Property Pages* in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

7.6.1. 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 , refer to Using Directives.

The vectorizer performs the following operations:

- Loop interchange
- Loop splitting

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

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

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

where **c** is the actual cache size of the machine.

This enables the options for nested loop transformation and various other vectorizer options. These defaults may vary depending on the target system. The following table lists and briefly describes some of the -Mvect suboptions.

Table 11 -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 SSE instructions are not generated.
-Mvect=short	Enable short vector operations.

Use this option ...	To instruct the vectorizer to do this ...
-Mvect=simd	Automatically generate packed SSE (Streaming SIMD Extensions), and prefetch instructions when vectorizable loops are encountered. SIMD instructions, first introduced on Pentium III and AthlonXP processors, operate on single-precision floating-point data.
-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 SSE (or SIMD) instructions on x86, compile with -Mvect=nosimd.

7.6.2. Vectorization Example Using SIMD Instructions

One of the most important vectorization options is -Mvect=simd. When you use this option, the compiler automatically generates SSE instructions, where possible, when targeting x86 processors on which these instructions are supported. This process can improve performance by several factors compared with the equivalent scalar code. All of the PGI Fortran, C and C++ compilers support this capability. The PGI Release Notes, www.pgroup.com/resources/docs/18.5/pdf/pgirn185-x86.pdf show which x86 and x64 processors PGI supports.

In the program in Vector operation using SIMD instructions, the vectorizer recognizes the vector operation in subroutine 'loop' when either the compiler switch -Mvect=simd or -fast is used. This example shows the compilation, informational messages, and runtime results using the SSE instructions on a x86 4 Core Intel Sandybridge 2.5 GHz system, along with issues that affect SSE performance.

Loops vectorized using x86 SSE instructions operate much more efficiently when processing vectors that are aligned to a cache-line boundary. You can cause unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the -Mcache_align switch. An unconstrained data object is a data object that is not a common block member and not a member of an aggregate data structure.

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 x86 SSE vectorization:

```
% pgfortran -fast -Mvect=nosimd -Minfo vadd.f -Mfree -o vadd
 vector_op:
 4, Loop unrolled 16 times
    Generates 1 prefetches in scalar loop
 9, Loop not vectorized/parallelized: contains call
 loop:
 18, Loop unrolled 4 times
```

The following output shows a sample result if the generated executable is run and timed on a x86 4 Core Intel Sandybridge 2.5 GHz system:

```
% /bin/time vadd
 -1.000000 -771.000 -3618.000 -6498.00 -9999.00
 1.35user 0.00system 0:01.35elapsed 99%CPU (0avgtext+0avgdata 3936maxresident)k
0inputs+0outputs (0major+290minor)pagefaults 0swaps
```

Now, recompile with x86 SSE 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:
     17, Generated 4 alternate versions of the loop
         Generated vector sse code for the loop
         Generated 3 prefetch instructions for the loop
```

Notice the informational message for the loop at line 17.

- The first two lines of the message indicate that the loop was vectorized, x86 SSE instructions were generated, and four alternate versions of the loop were also

generated. The loop count and alignments of the arrays determine which of these versions is executed.
- The last line of the informational message indicates that prefetch instructions have been generated for three loads to minimize latency of data transfers from main memory.

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

```
% /bin/time vadd
-1.000000 -771.000 -3618.00 -6498.00 -9999.0
0.60user 0.00system 0:00.61elapsed 99%CPU (0avgtext+0avgdata 3920maxresident)k
0inputs+0outputs (0major+289minor)pagefaults 0swaps
```

The SIMD result is 2.25 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 vector x86 SSE or SSE2 instructions is most effective.
- If data is aligned properly, performance will be better in general than when using vector x86 SSE operations on unaligned data.
- If the compiler can guarantee that data is aligned properly, even more efficient sequences of x86 SSE instructions can be generated.
- The efficiency of loops that operate on single-precision data can be higher. x86 SSE2 vector 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).

7.7. 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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

In PVF, the basic form of this option is accessed using the Auto-Parallelization property of the Fortran | Optimization property page. For more advanced auto-parallelization, use the Fortran | Command Line property page. For more information on these property pages, refer to 'Fortran Property Pages' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.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.

7.7.1. 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. For details on the use of directives, refer to Using Directives.

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 12 -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,

Use this option ...	To instruct the parallelizer to do this...
	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.

7.7.2. 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 loop:

```
! 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 example.

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

```
! 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 to let the compiler know the loop is safe to parallelize. The directive `safe_lastval` informs the compiler that, for a given loop, all scalars are assigned in the last iteration of the loop; thus, it is safe to parallelize the loop. We could add the following line to any of our previous examples.

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

The resulting code looks similar to this:

```
! 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.

7.8. Processor-Specific Optimization and the Unified Binary

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

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

You can use the `-tp` option to control compilation behavior by specifying the processor or processors with which the generated code is compatible. The compilers generate and combine into one executable multiple binary code streams, each optimized for a specific platform. At runtime, the executable senses the environment and dynamically selects the appropriate code stream. For specific information on the `-tp` option, refer to the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

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

7.9. Profile-Feedback Optimization using -Mpfi/-Mpfo

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

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

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

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

7.10. 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 13 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.

7.11. Local Optimization Using Directives

Command-line options let you specify optimizations for an entire source file. Directives supplied within a Fortran 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 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 provides details on how to add directives and pragmas to your source files.

7.12. Execution Timing and Instruction Counting

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

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

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

Timing functions available with the PGI compilers include these:

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

Chapter 8. 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`

8.1. 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.

In PVF, inlining can be turned on using the Inlining property in the Fortran | Optimization property page. For more advanced configuration of inlining, use the Fortran | Command Line property page. For more information on these property pages, refer to 'Fortran Property Pages' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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 `myprog.exe`.

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

For more information on the `-Minline` options, refer to '-M Options by Category' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

8.2. 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 `myprog.exe`.

```
$ 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
```

8.3. 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.

8.3.1. 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.

8.3.2. 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 or a PVF property and ensures that the necessary compilations are performed when a library is changed.

8.3.3. 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
```

8.4. 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
```

8.5. 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
```

8.6. 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.

Chapter 9. USING OPENMP

The PGF77 and PGFORTRAN Fortran compilers support the OpenMP Fortran 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 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.

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

9.1. OpenMP Overview

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

9.1.1. OpenMP Shared-Memory Parallel Programming Model

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

Fortran directives

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.

Fixed-form Fortran OpenMP directives begin with `!$OMP`, `C$OMP`, or `*$OMP`, beginning in column 1. Free-form Fortran OpenMP pragmas begin with `!$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 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 with additional directives.

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`.

9.1.2. 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.

9.1.3. 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
```

9.2. 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.

9.3. 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 Summary Table. The valid clauses depend on the directive. Directive Clauses provides a list of clauses, the directives to which they apply, and their functionality.

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

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

Clauses associated with directives have these characteristics:

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

9.4. Directive Recognition

The compiler option `–mp` enables recognition of the parallelization directives.

The use of this option also implies:

`-Miomutex`

For directives, critical sections are generated around Fortran I/O statements.

In PVF, you set the `––mp` option by using the Enable OpenMP Directives property in the Fortran | Language property page. For more information on these property pages, refer to the 'Fortran Property Pages' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

9.5. Directive Summary Table

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

9.5.1. Directive Summary Table

Table 14 Directive and Pragma Summary Table

Fortran Directive	Description
ATOMIC [TYPE} ... END ATOMIC	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive. 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	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	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
DO...END DO	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	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	Designates code that executes on the master thread and that is skipped by the other threads.
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	Enables you to specify which loops the compiler should parallelize.
PARALLEL ... END PARALLEL	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
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	Defines a non-iterative work-sharing construct within a parallel region.
SINGLE ... END SINGLE	Designates code that executes on a single thread and that is skipped by the other threads.
TASK	Defines an explicit task.

Fortran Directive	Description
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	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
THREADPRIVATE	When a common block or variable that is initialized appears in this directive, 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.

9.6. Directive Clauses

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

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 15 Directive and Pragma Summary Table

This clause...	Applies to this directive	Has this functionality
'CAPTURE'	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	Specifies how many loops are associated with the loop construct.
'COPYIN (list)'	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	Allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region. Then, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.
'COPYPRIVATE(list)'	SINGLE	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.

This clause...	Applies to this directive	Has this functionality
'DEFAULT'	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.
'FINAL'	TASK	Specifies that all subtasks of this task will be run immediately.
'FIRSTPRIVATE(list)'	DO PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE 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	Specifies whether a loop should be executed in parallel or in serial.
'LASTPRIVATE(list)'	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	Specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration of a for-loop construct.
'MERGEABLE'	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	Eliminates the barrier implicit at the end of a parallel region.
'NUM_THREADS'	PARALLEL PARALLEL DO ... END PARALLEL DO	Sets the number of threads in a thread team.

This clause...	Applies to this directive	Has this functionality
	PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	
‘ORDERED’	DO...END DO PARALLEL DO... END PARALLEL DO	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	Specifies that each thread should have its own instance of a variable.
‘READ’	ATOMIC	Specifies that the atomic action is reading a value.

‘REDUCTION’
`({operator`
`| intrinsic }`
`:`
`list)`
	DO
	PARALLEL
	PARALLEL DO ...
	END PARALLEL DO
	PARALLEL SECTIONS ...
	END PARALLEL SECTIONS
	PARALLEL WORKSHARE
	SECTIONS
		Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.

‘SCHEDULE’
`(type[`
`,chunk])`
	DO ... END DO
	PARALLEL DO...
	END PARALLEL DO
		Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

‘SHARED’	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS	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

This clause...	Applies to this directive	Has this functionality
	PARALLEL WORKSHARE	
'UNTIED'	TASK TASKWAIT	Specifies that any thread in the team can resume the task region after a suspension.
'UPDATE'	ATOMIC	Specifies that the atomic action is updating a value.
'WRITE'	ATOMIC	Specifies that the atomic action is writing a value.

9.7. Runtime Library Routines

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

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.

Table 16 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()`
omp_set_num_threads	
Sets the number of threads to use for the next parallel region. This subroutine can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine that is called from within a parallel region, the results are undefined. Further, this subroutine has precedence over the `OMP_NUM_THREADS` environment variable.	
Fortran	`subroutine omp_set_num_threads(scalar_integer_exp)`
omp_get_thread_num	

Runtime Library Routines with Examples	
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()`
omp_get_ancestor_thread_num	
Returns, for a given nested level of the current thread, the thread number of the ancestor.	
Fortran	`integer function omp_get_ancestor_thread_num(level)` `    integer level`
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()`
omp_get_level	
Returns the number of parallel regions enclosing the task that contains the call.	
Fortran	`integer function omp_get_level()`
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()`
omp_get_num_procs	
Returns the number of processors that are available to the program	
Fortran	`integer function omp_get_num_procs()`
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`

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))`
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`
omp_in_final	
Returns whether or not the call is within a final task. Returns `.TRUE.` if called from within a parallel region and `.FALSE.` 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.`, the function returns `.FALSE.`.	
Fortran	`integer function omp_in_final()`
omp_in_parallel	
Returns whether or not the call is within a parallel region. Returns `.TRUE.` if called from within a parallel region and `.FALSE.` 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.`, the function returns `.FALSE.`.	
Fortran	`logical function omp_in_parallel()`
omp_set_dynamic	
Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_dynamic(scalar_logical_exp)`
omp_get_dynamic	
Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.` for directives and zero for pragmas. This function is recognized, but currently always returns `.FALSE.`.	
Fortran	`logical function omp_get_dynamic()`
omp_set_nested	
Allows enabling/disabling of nested parallel regions.	
Fortran	`subroutine omp_set_nested(nested)` `logical nested`
omp_get_nested	

Runtime Library Routines with Examples	
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()`
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`

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`

omp_get_wtime	
Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value. Times returned are per-thread times, and are not necessarily globally consistent across all threads.	
Fortran	`double precision function omp_get_wtime()`
omp_get_wtick	
Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value.	
Fortran	`double precision function 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'`
	`   integer(kind=omp_lock_kind) 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`

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'`

<table>
<tr><th colspan="2">Runtime Library Routines with Examples</th></tr>
<tr><td></td><td><pre>   integer(kind=omp_lock_kind) lock</pre></td></tr>
<tr><td colspan="2">omp_unset_lock</td></tr>
<tr><td colspan="2">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.</td></tr>
<tr><td>Fortran</td><td><pre>subroutine omp_unset_lock(lock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_lock_kind) lock</pre></td></tr>
<tr><td colspan="2">omp_test_lock</td></tr>
<tr><td colspan="2">Causes the calling thread to try to gain ownership of the lock associated with the variable.
The function returns <code>.TRUE.</code> if the thread gains ownership of the lock; otherwise, it returns <code>.FALSE.</code>.
If the variable is not already associated with a lock, it is illegal to make a call to this routine.</td></tr>
<tr><td>Fortran</td><td><pre>logical function omp_test_lock(lock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_lock_kind) lock</pre></td></tr>
</table>

9.8. 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.

> To set the environment for programs run from within PVF, whether or not they are run in the debugger, use the environment properties available in the 'Debugging Property Page' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

Table 17 OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_MAX_ACTIVE_LEVELS	1	Specifies the maximum number of nested parallel regions.
OMP_NESTED	FALSE	Enables (TRUE) or disables (FALSE) nested parallelism.
OMP_NUM_THREADS	1	Specifies the number of threads to use during execution of parallel regions at the corresponding nested level. For example, OMP_NUM_THREADS=4,2 uses 4 threads at

Environment Variable	Default	Description
		the first nested parallel level, and 2 at the next nested parallel level.
OMP_SCHEDULE	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for *omp for* and *omp parallel for* loops that include the runtime schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
OMP_PROC_BIND	FALSE	Specifies whether executing threads should be bound to a core during execution. Allowable values are "true" and "false".
OMP_STACKSIZE		Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	64	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

Chapter 10. 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 that can be offloaded from a *host* CPU to an attached *accelerator*.

10.1. 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 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.

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.

10.1.1. 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.

10.1.2. 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.

10.2. 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
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
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.

10.3. 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.

10.3.1. 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.

10.3.2. 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.

10.4. 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.

10.4.1. 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.

10.4.2. 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.

10.4.3. 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.

10.4.4. CUDA Unified Memory

10.5. 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.

10.5.1. Enable Accelerator Directives

PGI compilers enable accelerator directives with the `-acc` and `-ta` command line options.In PVF, use the 'Fortran | Target Accelerators' page to enable the `-ta` option and the 'Fortran | Language' page to enable the `-acc` option. 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.

10.5.2. 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

10.5.3. 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.

10.6. Supported Processors and GPUs

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

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.

In PVF you can use the PVF Target Accelerators property page to enable accelerator compilation. For more information on the properties, refer to 'Fortran | Target Accelerators' section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

10.7. CUDA Toolkit Versions

The PGI compilers use NVIDIA's CUDA Toolkit when building programs for execution on an NVIDIA GPU. Every PGI installation packages puts the required CUDA Toolkit components into a PGI installation directory called `2018/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. Use it if you are unsure which version of the CUDA Driver is installed on your system.

PGI 18.5 contains the following versions of the CUDA Toolkits:

- CUDA 8.0 (default)
- CUDA 9.0
- CUDA 9.1
- CUDA 9.2

By default, the PGI compilers in this release use the CUDA 8.0 Toolkit from the PGI installation directory. You can compile with a different version of the CUDA Toolkit using one of the following methods:

- Use a compiler option. 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.

By default, the PGI compilers use the CUDA Toolkit components installed with the PGI compilers and in fact most users do not need to use any other CUDA Toolkit installation than those provided with PGI. 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.

The ability to compile with a CUDA Toolkit other than the versions installed with the PGI compilers is a feature not currently supported on Windows.

To use a CUDA toolkit that is not installed with a PGI release, such as CUDA 7.5 with PGI 18.5, there are three options:

- Use the rcfile variable `DEFAULT_CUDA_HOME` to override the base default

  ```
  set DEFAULT_CUDA_HOME = /opt/cuda-7.5;
  ```
- Set the environment variable `CUDA_HOME`

  ```
  export CUDA_HOME=/opt/cuda-7.5
  ```
- Use the compiler compilation line assignment CUDA_HOME=

  ```
  pgfortran CUDA_HOME=/opt/cuda-7.5
  ```

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

- In the absence of any other specification, the CUDA Toolkit located in the PGI installation directory `2018/cuda` will be used.
- The rcfile variable `DEFAULT_CUDA_HOME` will override the base default.
- The environment variable `CUDA_HOME` will override all of the above defaults.
- 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 `2018/cuda` will be used.
- The compiler compilation line assignment `CUDA_HOME=` will override all of the above defaults (including the `cudaX.Y` sub-option).
- The environment variable `PGI_CUDA_HOME` overrides all of the above; reserve `PGI_CUDA_HOME` for advanced use.

10.8. Compiling an Accelerator Program

Several compiler options are applicable specifically when working with accelerators. These options include `-ta`, `-acc`, and `-Minfo`. Each of these command line options are available through PVF's property pages.

10.8.1. Applicable PVF Property Pages

The following property pages are applicable specifically when working with accelerators.

'Fortran | Target Accelerators'
: Use the `-ta` option to enable recognition of Accelerator directives.

'Fortran | Target Processors'
: Use the `-tp` option to specify the target host processor architecture.

'Fortran | Diagnostics'
: Use the `-Minfo` option to see messages about the success or failure of the compiler in translating the accelerator region into GPU kernels.

For more information about the many suboptions available with these options, refer to the respective sections in the 'Fortran Property Pages' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

10.8.2. -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:

cc20, cc30, cc35, cc50, cc60, cc70
Generate code for compute capability 2.0, 3.0, 3.5, 5.0, 6.0, or 7.0 respectively; multiple selections are valid

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

[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]llvm
Generate [do not generate] code using the llvm-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

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`.

10.8.3. -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 device.

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.

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
```

10.9. 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. 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.

10.10. Compute Capability

The default for OpenACC and CUDA Fortran compilation for NVIDIA Tesla targets is to generate code for compute capabilities 3.5 through 6.0. If CUDA 9.0, 9.1, or 9.2 is specified, the default will also include compute capability 7.0. 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. 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 (`mypgi_rc` on Windows) in your home directory.

10.11. Running an Accelerator Program

In PVF you can use the PVF Target Accelerators property page to enable accelerator compilation. For more information on the properties, refer to the 'Fortran | Target Accelerators' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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. If the libraries are not available, or if they are in a different directory than they were when the program was compiled, you may need to append the appropriate library directory to your PATH environment variable on Windows.
- On Linux, if you have no server running on your NVIDIA GPU, when your program reaches its first accelerator region, there may be a 0.5 to 1.5 second pause to warm up the GPU from a power-off 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.

10.12. 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 18 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 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.

10.13. 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.

> Windows Users: To ensure that all the performance information is collected we recommend that 'acc_shutdown' is called before your application is finished and 'main' exits.

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.

10.14. 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.

10.14.1. Runtime Library Definitions

There are separate runtime library files for Fortran.

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.

10.14.2. Runtime Library Routines

Table 19 lists and briefly describes the runtime library routines supported by PGI in addition to the standard OpenACC runtine API routines.

Table 19 Accelerator Runtime Library Routines

This Runtime Library Routine...	Does this...
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.

10.15. 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 intrinsics that the accelerator supports.

10.15.1. Supported Fortran Intrinsics Summary Table

Table 20 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/18.5/pdf/pgi18fortref-x86.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 20 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.

Chapter 11.
USING DIRECTIVES

It is often useful to be able to alter the effects of certain command line options or default behavior of the compiler. Fortran directives 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 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 to tune selected routines or loops.

11.1. 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$`.

11.2. PGI Proprietary Optimization Directive Summary

The following table summarizes the supported Fortran directives. The following terms are useful in understanding the table.

- Functionality is a brief summary of the way to use the directive. For a complete description, refer to the 'Directives Reference' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.
- Many of the directives can be preceded by `NO`. The default entry indicates the default for the directive. N/A appears if a default does not apply.
- The scope entry indicates the allowed scope indicators for each directive, with `l` for loop, `r` for routine, and `g` for global. The default scope is surrounded by parentheses.

> The "*" in the scope indicates this:
> **For routine-scoped directive**
> The scope includes the code following the directive until the end of the routine.
> **For globally-scoped directive**
> The scope includes the code following the directive until the end of the file rather than for the entire file.

> The name of a directive 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`.

Table 21 Proprietary Optimization-Related Fortran Directive Summary

Directive	Functionality	Default	Scope
altcode (noaltcode)	Do/don't generate alternate code for vectorized and parallelized loops.	altcode	(l)rg
assoc (noassoc)	Do/don't perform associative transformations.	assoc	(l)rg
bounds (nobounds)	Do/don't perform array bounds checking.	nobounds	(r)g*

Directive	Functionality	Default	Scope
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
concur (noconcur)	Do/don't enable auto-concurrentization of loops.	concur	(l)rg
depchk (nodepchk)	Do/don't ignore potential data dependencies.	depchk	(l)rg
eqvchk (noeqvchk)	Do/don't check EQUIVALENCE s for data dependencies.	eqvchk	(l)rg
invarif (noinvarif)	Do/don't remove invariant if constructs from loops.	invarif	(l)rg
ivdep	Ignore potential data dependencies.	ivdep	(l)rg
lstval (nolstval)	Do/don't compute last values.	lstval	(l)rg
prefetch	Control how prefetch instructions are emitted		
opt	Select optimization level.	N/A	(r)g
safe_lastval	Parallelize when loop contains a scalar used outside of loop.	not enabled	(l)
tp	Generate PGI Unified Binary code optimized for specified targets.	N/A	(r)g
unroll (nounroll)	Do/don't unroll loops.	nounroll	(l)rg
vector (novector)	Do/don't perform vectorizations.	vector	(l)rg*
vintr (novintr)	Do/don't recognize vector intrinsics.	vintr	(l)rg

11.3. 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.

11.4. 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.

For a list of processors that support prefetch instructions refer to the PGI Release Notes.

11.4.1. 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.

11.4.2. 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.

11.4.3. 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.

11.5. 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.

11.5.1. 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.

11.5.2. 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.

11.5.3. 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 22 indicates which rules are ignored for which dummy arguments in the preceding sample subroutine fragment:

Table 22 IGNORE_TKR Example

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

Dummy Argument	Ignored Rules
D	Only Kind

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

11.6. !DEC$ Directives

PGI Fortran compilers for Microsoft Windows support several de-facto standard Fortran directives that help with inter-language calling and importing and exporting routines to and from DLLs.

11.6.1. !DEC$ Directive Syntax

These directives all take the form:

```
!DEC$ directive
```

11.6.2. Format Requirements

You must follow the following format requirements for the directive to be recognized in your program:

- The directive must begin in column 1 when the file is fixed format or compiled with -Mfixed.
- The directive prefix **`!DEC$`** requires a space between the prefix and the directive keyword, such as **`ATTRIBUTES`**.
- The **`!`** must begin the prefix when compiling Fortran 90/95 free-form format.
- The characters **`C`** or **`*`** can be used in place of **`!`** in either form of the prefix when compiling F77-style fixed-form format.
- The directives are completely case insensitive.

11.6.3. Summary Table

The following table summarizes the supported !DEC$ directives. For a complete description of each directive, refer to the '!DEC$ Directives' section of the 'Directives and Pragmas Reference' section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

Table 23 !DEC$ Directives Summary Table

Directive	Functionality
ALIAS	Specifies an alternative name with which to resolve a routine.
ATTRIBUTES	Lets you specify properties for data objects and procedures.
DECORATE	Specifies that the name specified in the ALIAS directive should have the prefix and postfix decorations performed on it that are associated with the calling conventions that are in effect. This directive has no effect if ALIAS is not specified.

Directive	Functionality
DISTRIBUTE	Tells the compiler at what point within a loop to split into two loops.

Chapter 12. 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 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. On Windows, PGI also provides additions to the supported library functionality for runtime functions included in DFLIB.

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

12.1. PGI Runtime Libraries on Windows

Both statically- and dynamically-linked library (DLL) versions are available with the PGI runtime libraries on Windows. The static libraries are used by default.

- You can use the dynamically-linked version of the runtime by specifying `-Bdynamic` at both compile and link time.
- You can explicitly specify static linking, the default, by using `-Bstatic` at compile and link time.

For details on why you might choose one type of linking over another type, refer to Creating and Using Dynamic-Link Libraries on Windows.

12.2. Creating and Using Static Libraries on Windows

The Microsoft Library Manager (`LIB.EXE`) is the tool that is typically used to create and manage a static library of object files on Windows. `LIB` is provided with the PGI compilers as part of the Microsoft Open Tools. Refer to http://www.msdn2.com for a complete `LIB` reference – search for `LIB.EXE`. For a list of available options, invoke `LIB` with the `/?` switch.

For compatibility with legacy makefiles, PGI provides a wrapper for `LIB` and `LINK` called `ar`. This version of `ar` is compatible with Windows and object-file formats.

PGI also provides `ranlib` as a placeholder for legacy makefile support.

12.2.1. ar command

The `ar` command is a legacy archive wrapper that interprets legacy `ar` command line options and translates these to `LINK/LIB` options. You can use it to create libraries of object files.

Syntax

The syntax for the `ar` command is this:

```
ar [options] [archive] [object file].
```

Where:

- The first argument must be a command line switch, and the leading dash on the first option is optional.
- The single character options, such as `-d` and `-v`, may be combined into one option, such as `-dv`.

  Thus, `ar dv`, `ar -dv`, and `ar -d -v` all mean the same thing.
- The first non-switch argument must be the library name.
- Exactly one of `-d`, `-r`, `-t`, or `-x` must appear on the command line.

Options

The options available for the `ar` command are these:

-c

This switch is for compatibility; it is ignored.

-d

Deletes the named object files from the library.

-r

Replaces in or adds the named object files to the library.

-t
Writes a table of contents of the library to standard out.

-v
Writes a verbose file-by-file description of the making of the new library to standard out.

-x
Extracts the named files by copying them into the current directory.

12.2.2. ranlib command

The `ranlib` command is a wrapper that allows use of legacy scripts and makefiles that use the `ranlib` command. The command actually does nothing; it merely exists for compatibility.

Syntax

The syntax for the `ranlib` command is this:

```
ranlib [options] [archive]
```

Options

The options available for the `ar` command are these:

-help
Short help information is printed out.

-V
Version information is printed out.

12.3. Creating and Using Dynamic-Link Libraries on Windows

There are several differences between static- and dynamic-link libraries on Windows. Libraries of either type are used when resolving external references for linking an executable, but the process differs for each type of library. When linking with a static library, the code needed from the library is incorporated into the executable. When linking with a DLL, external references are resolved using the DLL's import library, not the DLL itself. The code in the DLL associated with the external references does not become a part of the executable. The DLL is loaded when the executable that needs it is run. For the DLL to be loaded in this manner, the DLL must be in your path.

Static libraries and DLLs also handle global data differently. Global data in static libraries is automatically accessible to other objects linked into an executable. Global data in a DLL can only be accessed from outside the DLL if the DLL exports the data and the image that uses the data imports it.

The PGI Fortran compilers support the DEC$ ATTRIBUTES extensions `DLLIMPORT` and `DLLEXPORT`:

```
cDEC$ ATTRIBUTES DLLEXPORT :: object [,object] ...
cDEC$ ATTRIBUTES DLLIMPORT :: object [,object] ...
```

Here *c* is one of C, c, !, or *. `object` is the name of the subprogram or common block that is exported or imported. Further, common block names are enclosed within slashes (/), as shown here:

```
cDEC$ ATTRIBUTES DLLIMPORT :: intfunc
!DEC$ ATTRIBUTES DLLEXPORT :: /fdata/
```

For more information on these extensions, refer to !DEC$ Directives.

The examples in this section further illustrate the use of these extensions.

To create a DLL in PVF, select *File* | *New* | *Project...*, then select *PGI Visual Fortran*, and create a new Dynamic Library project.

To create a DLL from the command line, use the `-Mmakedll` option.

The following switches apply to making and using DLLs with the PGI compilers:

-Bdynamic
: Compile for and link to the DLL version of the PGI runtime libraries. This flag is required when linking with any DLL built by the PGI compilers. This flag corresponds to the `/MD` flag used by Microsoft's `cl` compilers.

  When you use the PGI compiler flag `-Bdynamic` to create an executable that links to the DLL form of the runtime, the executable built is smaller than one built without `-Bdynamic`. The PGI runtime DLLs, however, must be available on the system where the executable is run. You must use the `-Bdynamic` flag when linking an executable against a DLL built by the PGI compilers.

-Bstatic
: Compile for and link to the static version of the PGI runtime libraries. This flag corresponds to the `/MT` flag used by Microsoft's `cl` compilers.

  On Windows, you must use`-Bstatic` for both compiling and linking.

-Mmakedll
: Generate a dynamic-link library or DLL. Implies `-Bdynamic`.

-Mmakeimplib
: Generate an import library without generating a DLL. Use this flag when you want to generate an import library for a DLL but are not yet ready to build the DLL itself. This situation might arise, for example, when building DLLs with mutual imports, as shown in Build DLLs Containing Mutual Imports: Fortran.

-o <file>
: Passed to the linker. Name the DLL or import library <file>.

-def <file>
: When used with `-Mmakedll`, this flag is passed to the linker and a `.def` file named <file> is generated for the DLL. The `.def` file contains the symbols exported by the DLL. Generating a `.def` file is not required when building a DLL but can be a useful debugging tool if the DLL does not contain the symbols that you expect it to contain.

  When used with `-Mmakeimplib`, this flag is passed to `lib` which requires a `.def` file to create an import library. The `.def` file can be empty if the list of symbols to export are passed to `lib` on the command line or explicitly marked as `DLLEXPORT` in the source code.

-implib <file>
Passed to the colinker. Generate an import library named <file> for the DLL. A DLL's import library is the interface used when linking an executable that depends on routines in a DLL.

To use the PGI compilers to create an executable that links to the DLL form of the runtime, use the compiler flag `-Bdynamic`. The executable built will be smaller than one built without `-Bdynamic`; the PGI runtime DLLs, however, must be available on the system where the executable is run. The `-Bdynamic` flag must be used when an executable is linked against a DLL built by the PGI compilers.

The following examples outline how to use `-Bdynamic`, `-Mmakedll` and `-Mmakeimplib` to build and use DLLs with the PGI compilers.

12.3.1. Build a DLL: Fortran

This example builds a DLL from a single source file, `object1.f`, which exports data and a subroutine using `DLLEXPORT`. The source file, `prog1.f`, uses `DLLIMPORT` to import the data and subroutine from the DLL.

`object1.f`

```
 subroutine sub1(i)
!DEC$ ATTRIBUTES DLLEXPORT :: sub1
 integer i
 common /acommon/ adata
 integer adata
!DEC$ ATTRIBUTES DLLEXPORT :: /acommon/
 print *, "sub1 adata", adata
 print *, "sub1 i ", i
 adata = i
 end
```

`prog1.f`

```
 program prog1
 common /acommon/ adata
 integer adata
 external sub1
!DEC$ ATTRIBUTES DLLIMPORT:: sub1, /acommon/
 adata = 11
 call sub1(12)
 print *, "main adata", adata
 end
```

1. Create the DLL `obj1.dll` and its import library `obj1.lib` using the following series of commands:

   ```
   % pgfortran -Bdynamic -c object1.f
   % pgfortran -Mmakedll object1.obj -o obj1.dll
   ```

2. Compile the main program:

   ```
   % pgfortran -Bdynamic -o prog1 prog1.f -defaultlib:obj1
   ```

   The `-Bdynamic` and `-Mmakedll` switches cause the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `-Bdynamic` switch is required when linking against any PGI-compiled DLL, such as `obj1.dll`. The -defaultlib: switch specifies that `obj1.lib`, the DLL's import library, should be used to resolve imports.

3. Ensure that `obj1.dll` is in your path, then run the executable `prog1` to determine if the DLL was successfully created and linked:

```
% prog1
sub1 adata 11
sub1 i 12
main adata 12
```

Should you wish to change `obj1.dll` without changing the subroutine or function interfaces, no rebuilding of `prog1` is necessary. Just recreate `obj1.dll` and the new `obj1.dll` is loaded at runtime.

12.3.2. Build DLLs Containing Mutual Imports: Fortran

In this example we build two DLLs when each DLL is dependent on the other, and use them to build the main program.

In the following source files, `object2.f95` makes calls to routines defined in `object3.f95`, and vice versa. This situation of mutual imports requires two steps to build each DLL.

To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation.

Once the DLLs are built, we can use them to build the main program.

`object2.f95`

```
 subroutine func_2a
 external func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3b
   print*,"func_2a, calling a routine in obj3.dll"
 call func_3b()  end subroutine
```

```
 subroutine func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2b
  print*,"func_2b"
end subroutine
```

`object3.f95`

```
 subroutine func_3a
   external func_2b
   !DEC$ ATTRIBUTES DLLEXPORT :: func_3a
   !DEC$ ATTRIBUTES DLLIMPORT :: func_2b
   print*,"func_3a, calling a routine in obj2.dll"
   call func_2b()  end subroutine
```

```
 subroutine func_3b
 !DEC$ ATTRIBUTES DLLEXPORT :: func_3b
    print*,"func_3b"
```

```
end subroutine
```

`prog2.f95`

```
program prog2
    external func_2a
    external func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3a
    call func_2a()
    call func_3a()
end program
```

1. Use `–Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

   ```
   % pgfortran -Bdynamic -c object2.f95
   % pgfortran -Mmakeimplib -o obj2.lib object2.obj
   ```

   > **Tip** The `–def=<deffile>` option can also be used with `–Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `DLLEXPORT`.

2. Use the import library, `obj2.lib`, created in Step 1, to link the second DLL.

   ```
   % pgfortran -Bdynamic -c object3.f95
   % pgfortran -Mmakedll -o obj3.dll object3.obj -defaultlib:obj2
   ```

3. Use the import library, `obj3.lib`, created in Step 2, to link the first DLL.

   ```
   % pgfortran -Mmakedll -o obj2.dll object2.obj -defaultlib:obj3
   ```

4. Compile the main program and link against the import libraries for the two DLLs.

   ```
   % pgfortran -Bdynamic prog2.f95 -o prog2 -defaultlib:obj2 -defaultlib:obj3
   ```

5. Execute `prog2` to ensure that the DLLs were create properly.

   ```
   % prog2
   func_2a, calling a routine in obj3.dll
   func_3b
   func_3a, calling a routine in obj2.dll
   func_2b
   ```

12.3.3. Import a Fortran module from a DLL

In this example we import a Fortran module from a DLL. We use the source file `defmod.f90` to create a DLL containing a Fortran module. We then use the source file `use_mod.f90` to build a program that imports and uses the Fortran module from `defmod.f90`.

defmod.f90

```
module testm
 type a_type
 integer :: an_int
 end type a_type
 type(a_type) :: a, b
!DEC$ ATTRIBUTES DLLEXPORT :: a,b
 contains
 subroutine print_a
!DEC$ ATTRIBUTES DLLEXPORT :: print_a
 write(*,*) a%an_int
 end subroutine
 subroutine print_b
```

```
!DEC$ ATTRIBUTES DLLEXPORT :: print_b
 write(*,*) b%an_int
 end subroutine
end module
```

usemod.f90

```
use testm
 a%an_int = 1
 b%an_int = 2
 call print_a
 call print_b
end
```

1. Create the DLL.

   ```
   % pgf90 -Mmakedll -o defmod.dll defmod.f90
   Creating library defmod.lib and object defmod.exp
   ```
2. Create the `exe` and link against the import library for the imported DLL.

   ```
   % pgf90 -Bdynamic -o usemod usemod.f90 -defaultlib:defmod.lib
   ```
3. Run the `exe` to ensure that the module was imported from the DLL properly.

   ```
   % usemod
   1
   2
   ```

12.4. Using LIB3F

The PGI Fortran compilers include complete support for the de facto standard LIB3F library routines on both Linux and Windows operating systems. See the PGI Fortran Language Reference manual for a complete list of available routines in the PGI implementation of LIB3F.

12.5. 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.lib`. The BLAS library is called `libblas.lib`. These libraries are installed to `$PGI\win64\18.5\lib`.

To use these libraries, simply link them in using the Linker option in the project's Configuration Properties. Either setting the Additional Dependencies via the Input sub-option, or adding `-llapack -lblas` to the Command Line sub-option works.

12.6. 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/linux86-64/18.5/lib/libblas.a`.

Chapter 13. 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/18.5/pdf/pgi18ref-x86.pdf, the PGI Debugger User's Guide, www.pgroup.com/resources/docs/18.5/pdf/pgi18dbug.pdf and the Profiler User's Guide, www.pgroup.com/resources/docs/18.5/pdf/pgi18profug.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 debugger or PGI profiler. For a description of environment variables that affect these tools, refer to the PGI Debugger User's Guide, www.pgroup.com/resources/docs/18.5/pdf/pgi18dbug.pdf and Profiler User's Guide, www.pgroup.com/resources/docs/18.5/pdf/pgi18profug.pdf, respectively.

13.1. 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.

13.1.1. Setting Environment Variables on Windows

When you open the PVF Command Prompt, as described in Commands Submenu, the environment is pre-initialized to use the PGI compilers and tools.

You may want to use other environment variables, such as the OpenMP ones. This section explains how to do that.

> To set the environment for programs run from within PVF, whether or not they are run in the debugger, use the environment properties described in the 'Debugging Property Page' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

Suppose that your home directory is `C:\tmp`. The following example shows how you might set the temporary directory to your home directory, and then verify that it is set.

```
DOS> set TMPDIR=C:\tmp
DOS> echo %TMPDIR%
C:\tmp
DOS>
```

13.2. 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 17 and to the complete descriptions in 'OpenMP Environment Variables' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

Table 24 PGI-Related Environment Variable Summary

Environment Variable	Description
FLEXLM_BATCH	(Windows only) When set to 1, prevents interactive pop-ups from appearing by sending all licensing errors and warnings to standard out rather than to a pop-up window.
FORTRANOPT	Allows the user to specify that the PGI Fortran compilers user VAX I/O conventions.
LM_LICENSE_FILE	Specifies the full path of the license file that is required for running the PGI software. On Windows, `LM_LICENSE_FILE` does not need to be set.
MPSTKZ	Increases the size of the stacks used by threads executing in parallel regions. The value should be an integer <n> concatenated with `M` or `m` to specify stack sizes of n megabytes.
MP_BIND	Specifies whether to bind processes or threads executing in a parallel region to a physical processor.
MP_BLIST	When `MP_BIND` is `yes`, this variable specifically defines the thread-CPU relationship, overriding the default values.
MP_SPIN	Specifies the number of times to check a semaphore before calling _sleep().
MP_WARN	Allows you to eliminate certain default warning messages.
NCPUS	Sets the number of processes or threads used in parallel regions.

Environment Variable	Description
NCPUS_MAX	Limits the maximum number of processors or threads that can be used in a parallel region.
NO_STOP_MESSAGE	If used, the execution of a plain STOP statement does not produce the message `FORTRAN STOP`.
OMP_DYNAMIC	Currently has no effect. Enables (`TRUE`) or disables (`FALSE`) the dynamic adjustment of the number of threads. The default is FALSE.
OMP_MAX_ACTIVE_LEVELS	Specifies the maximum number of nested parallel regions.
OMP_NESTED	Currently has no effect. Enables (`TRUE`) or disables (`FALSE`) nested parallelism. The default is FALSE.
OMP_NUM_THREADS	Specifies the number of threads to use during execution of parallel regions. Default is 1.
OMP_SCHEDULE	Specifies the type of iteration scheduling and, optionally, the chunk size to use for *omp for* and *omp parallel for* loops that include the runtime schedule clause. The default is STATIC with chunk size=1.
OMP_STACKSIZE	Overrides the default stack size for a newly created thread.
OMP_WAIT_POLICY	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are `ACTIVE` and `PASSIVE`. The default is `ACTIVE`.
PATH	Determines which locations are searched for commands the user may type.
PGI	Specifies, at compile-time, the root directory where the PGI compilers and tools are installed.
PGI_CONTINUE	If set, when a program compiled with`−Mchkfpstk` is executed, the stack is automatically cleaned up and execution then continues.
PGI_OBJSUFFIX	(Windows only) Allows you to control the suffix on generated object files.
PGI_STACK_USAGE	(Windows only) Allows you to explicitly set stack properties for your program.
PGI_TERM	Controls the stack traceback and just-in-time debugging functionality.
PGI_TERM_DEBUG	Overrides the default behavior when `PGI_TERM` is set to `debug`.
PGROUPD_LICENSE_FILE	Specifies the location of the PGI license. This variable is set in the registry on Windows machines, and is specific to PGI products. On Windows, `PGROUPD_LICENSE_FILE` does not need to be set.
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.

13.3. PGI Environment Variables

You use the environment variables listed in Table 24 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.

13.3.1. FLEXLM_BATCH

By default, on Windows the license server creates interactive pop-up messages to issue warning and errors. You can use the environment variable `FLEXLM_BATCH` to prevent interactive pop-up windows. To do this, set the environment variable `FLEXLM_BATCH` to 1.

The following csh example prevents interactive pop-up messages for licensing warnings and errors:

```
% set FLEXLM_BATCH = 1;
```

13.3.2. FORTRANOPT

`FORTRANOPT` allows the user to adjust the behavior of the PGI Fortran compilers.

- If `FORTRANOPT` exists and contains the value `vaxio`, the record length in the open statement is in units of 4-byte words, and the $ edit descriptor only has an effect for lines beginning with a space or a plus sign (+).
- If `FORTRANOPT` exists and contains the value `format_relaxed`, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) is not required to be a type implied by the descriptor.
- In a non-Windows environment, if `FORTRANOPT` exists and contains the value `crif`, a sequential formatted or list-directed record is allowed to be terminated with the character sequence `\r\n` (carriage return, newline). This approach is useful when reading records from a file produced on a Window's system.

The following example causes the PGI Fortran compilers to use VAX I/O conventions:

```
% setenv FORTRANOPT vaxio
```

13.3.3. LM_LICENSE_FILE

The `LM_LICENSE_FILE` variable specifies the full path of the license file that is required for running the PGI software.

> `LM_LICENSE_FILE` is not required for PVF, but you can use it.

To set the environment variable `LM_LICENSE_FILE` to the full path of the license key file, do this:

1. Open the System Properties dialog: *Start* | *Control Panel* | *System*.

2. Select the *Advanced* tab.
3. Click the *Environment Variables* button.
    - If LM_LICENSE_FILE is not already an environment variable, create a new system variable for it. Set its value to the full path, including the name of the license key file, license.dat.
    - If LM_LICENSE_FILE already exists as an environment variable, append the path to the license file to the variable's current value using a semi-colon to separate entries.

13.3.4. MPSTKZ

MPSTKZ increases the size of the stacks used by threads executing in parallel regions. You typically use this variable with programs that utilize large amounts of thread-local storage in the form of private variables or local variables in functions or subroutines called within parallel regions. The value should be an integer <n> concatenated with M or m to specify stack sizes of n megabytes.

For example, the following setting specifies a stack size of 8 megabytes.

```
% setenv MPSTKZ 8M
```

13.3.5. MP_BIND

You can set MP_BIND to yes or y to bind processes or threads executing in a parallel region to physical processor. Set it to no or n to disable such binding. The default is to not bind processes to processors. This variable is an execution-time environment variable interpreted by the PGI runtime support libraries. It does not affect the behavior of the PGI compilers in any way.

> The MP_BIND environment variable is not supported on all platforms.

```
% setenv MP_BIND y
```

13.3.6. MP_BLIST

MP_BLIST allows you to specifically define the thread-CPU relationship.

> This variable is only in effect when MP_BIND is yes.

While the MP_BIND variable binds processors or threads to a physical processor, MP_BLIST allows you to specifically define which thread is associated with which processor. The list defines the processor-thread relationship order, beginning with thread 0. This list overrides the default binding.

For example, the following setting for MP_BLIST maps CPUs 3, 2, 1 and 0 to threads 0, 1, 2 and 3 respectively.

```
% setenv MP_BLIST=3,2,1,0
```

13.3.7. MP_SPIN

When a thread executing in a parallel region enters a barrier, it spins on a semaphore. You can use `MP_SPIN` to specify the number of times it checks the semaphore before calling `_sleep()` . These calls cause the thread to be re-scheduled, allowing other processes to run. The default value is 10000.

```
% setenv MP_SPIN 200
```

13.3.8. MP_WARN

`MP_WARN` allows you to eliminate certain default warning messages.

By default, a warning is printed to standard error if you execute an OpenMP or auto-parallelized program with `NCPUS` or `OMP_NUM_THREADS` set to a value larger than the number of physical processors in the system.

For example, if you produce a parallelized executable `a.exe` and execute as follows on a system with only one processor, you get a warning message.

```
> set OMP_NUM_THREADS=2
> a.exe
Warning: OMP_NUM_THREADS or NCPUS (2) greater than available cpus (1)
FORTRAN STOP
```

Setting `MP_WARN` to `NO` eliminates these warning messages.

13.3.9. NCPUS

You can use the `NCPUS` environment variable to set the number of processes or threads used in parallel regions. The default is to use only one process or thread, which is known as serial mode.

> `OMP_NUM_THREADS` has the same functionality as `NCPUS`. For historical reasons, PGI supports the environment variable `NCPUS`. If both `OMP_NUM_THREADS` and `NCPUS` are set, the value of `OMP_NUM_THREADS` takes precedence.

Setting `NCPUS` to a value larger than the number of physical processors or cores in your system can cause parallel programs to run very slowly.

13.3.10. NCPUS_MAX

You can use the `NCPUS_MAX` environment variable to limit the maximum number of processes or threads used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using set_omp_num_threads(), will cause the number of processes or threads to be set at the value of `NCPUS_MAX` rather than the value specified in the function call.

13.3.11. 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.

13.3.12. 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.

```
% set path = (/opt/pgi/linux86-64/18.5/bin $path)
```

Within the PVF IDE, the `PATH` variable can be set using the Environment and MPI Debugging properties on the 'Debugging Property Page' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf. The PVF Command Prompt, accessible from the PVF submenus of *Start* | *All Programs* | *PGI Visual Fortran*, opens with the `PATH` variable pre-configured for use of the PGI products.

Important If you invoke a generic Command Prompt using *Start* | *All Programs* | *Accessories* | *Command Prompt*, then the environment is not pre-configured for PGI products.

13.3.13. 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 Windows, the default value is `C:\Program Files\PGI`, where `C` represents the system drive.

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.

```
% setenv PGI /opt/pgi
% setenv MANPATH "$MANPATH":$PGI/linux86-64/18.5/man
% setenv LM_LICENSE_FILE $PGI/license.dat
% set path = ($PGI/linux86-64/18.5/bin $path)
```

13.3.14. 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.

13.3.15. 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.

13.3.16. PGI_STACK_USAGE

(Windows only) The PGI_STACK_USAGE variable allows you to explicitly set stack properties for your program. When the user compiles a program with the -Mchkstk option and sets the PGI_STACK_USAGE environment variable to any value, the program displays the stack space allocated and used after the program exits. You might see something similar to the following message:

```
thread 0 stack: max 8180KB, used 48KB
```

This message indicates that the program used 48KB of a 8180KB allocated stack. For more information on the -Mchkstk option, refer to '-Mchkstk' in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

13.3.17. PGI_TERM

The PGI_TERM environment variable controls the stack traceback and just-in-time debugging functionality. The runtime libraries use the value of PGI_TERM to determine what action to take when a program abnormally terminates.

The value of PGI_TERM is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:
  ```
  % setenv PGI_TERM option[,option...]
  ```
- In bash, sh, zsh, or ksh:
  ```
  $ PGI_TERM=option[,option...]
  $ export PGI_TERM
  ```
- In the Windows Command Prompt:
  ```
  C:\> set PGI_TERM=option[,option...]
  ```

Table 25 lists the supported values for option. Following the table is a complete description of each option that indicates specifically how you might apply the option.

By default, all of these options are disabled.

Table 25 Supported PGI_TERM Values

[no]debug	Enables/disables just-in-time debugging (debugging invoked on error)
[no]trace	Enables/disables stack traceback on error

[no]signal	Enables/disables establishment of signal handlers for common signals that cause program termination
[no]abort	Enables/disables calling the system termination routine abort()

[no]debug

This enables/disables just-in-time debugging. The default is `nodebug`.

When `PGI_TERM` is set to `debug`, the following command is invoked on error, unless you use `PGI_TERM_DEBUG` to override this default.

```
pgdbg -text -attach <pid>
```

`<pid>` is the process ID of the process being debugged.

The PGI_TERM_DEBUG environment variable may be set to override the default setting. For more information, refer to PGI_TERM_DEBUG.

[no]trace

This enables/disables stack traceback on error.

[no]signal

This enables/disables establishing signal handlers for the most common signals that cause program termination. The default is `nosignal`. Setting `trace` and `debug` automatically enables `signal`. Specifically setting `nosignal` allows you to override this behavior.

[no]abort

This enables/disables calling the system termination routine abort(). The default is `noabort`. When `noabort` is in effect the process terminates by calling `_exit(127)`.

A few runtime errors just print an error message and call `exit(127)`, regardless of the status of `PGI_TERM`. These are mainly errors such as specifying an invalid environment variable value where a traceback would not be useful.

The abort routine exits with the status of the exception received; for example, if the program receives an access violation abort exits with status 0xC0000005.

For more information on why to use this variable, refer to Stack Traceback and JIT Debugging.

13.3.18. PGI_TERM_DEBUG

The `PGI_TERM_DEBUG` variable may be set to override the default behavior when `PGI_TERM` is set to `debug`.

The value of `PGI_TERM_DEBUG` should be set to the command line used to invoke the program. For example:

```
gdb --quiet --pid %d
```

The first occurrence of `%d` in the `PGI_TERM_DEBUG` string is replaced by the process id. The program named in the `PGI_TERM_DEBUG` string must be found on the current `PATH` or specified with a full path name.

13.3.19. PGROUPD_LICENSE_FILE

You can use the `PGROUPD_LICENSE_FILE` to specifies the location of the PGI license. This variable is set in the registry on Windows machines, and is specific to PGI products.

The system environment variable `PGROUPD_LICENSE_FILE` is not required by PGI products on Windows but you can use it to override the default location that is searched for the `license.dat` file.

To use the system environment variable `PGROUPD_LICENSE_FILE`, set it to the full path of the license keys file. To do this, follow these steps:

1. Open the System Properties dialog from Control Panel | System.
2. Select the 'Advanced' tab.
3. Click the 'Environment Variables' button.

    - If `PGROUPD_LICENSE_FILE` is not already an environment variable, create a new system variable for it. Set its value to the full path, including the name of the file, for the license keys file.
    - If `PGROUPD_LICENSE_FILE` already exists as an environment variable, append the path to the license file to the variable's current value using a semi-colon to separate entries.

13.3.20. 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.

13.3.21. 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`.

13.3.22. 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.

13.4. Stack Traceback and JIT Debugging

When a programming error results in a runtime error message or an application exception, a program will usually exit, perhaps with an error message. The PGI runtime library includes a mechanism to override this default action and instead print a stack tracebackor start a debugger.

The stack traceback and just-in-time debugging functionality is controlled by an environment variable, `PGI_TERM`, described in PGI_TERM. The runtime libraries use the value of `PGI_TERM` to determine what action to take when a program abnormally terminates.

When the PGI runtime library detects an error or catches a signal, it calls the routine `pgi_stop_here()` prior to generating a stack traceback or starting the debugger. The `pgi_stop_here()` routine is a convenient spot to set a breakpoint when debugging a program.

Chapter 14. 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.

14.1. Deploying Applications on Windows

Windows programs may be linked statically or dynamically.

- A statically linked program is completely self-contained, created by linking to static versions of the PGI and Microsoft runtime libraries.
- A dynamically linked program depends on separate dynamically-linked libraries (DLLs) that must be installed on a system for the application to run on that system.

Although it may be simpler to install a statically linked executable, there are advantages to using the DLL versions of the runtime, including:

- Executable binary file size is smaller.
- Multiple processes can use DLLs at once, saving system resources.
- New versions of the runtime can be installed and used by the application without rebuilding the application.

Dynamically-linked Windows programs built with PGI compilers depend on dynamic runtime library files (DLLs). These DLLs must be distributed with such programs to enable them to execute on systems where the PGI compilers are not installed. These redistributable libraries include both PGI runtime libraries and Microsoft runtime libraries.

14.1.1. PGI Redistributables

PGI redistributable directories contain all of the PGI Windows dynamically-linked libraries that can be re-distributed by PGI 18.5 licensees under the terms of the PGI End-User License Agreement (EULA).

14.1.2. Microsoft Redistributables

The PGI products on Windows include Microsoft Open Tools. The Microsoft Open Tools directory contains a subdirectory named `redist`. PGI licensees may redistribute the files contained in this directory in accordance with the terms of the End-User License Agreement.

Microsoft supplies installation packages, `vcredist_x86.exe` and `vcredist_x64.exe`, containing these runtime files. These files are available in the `redist` directory.

14.2. Code Generation and Processor Architecture

The PGI compilers can generate much more efficient code if they know the specific x86-64 processor architecture on which the program will run. When preparing to deploy your application, you should determine whether you want the application to run on the widest possible set of x86-64 processors, or if you want to restrict the application to run on a specific processor or set of processors. The restricted approach allows you to optimize performance for that set of processors.

Different processors have differences, some subtle, in hardware features, such as instruction sets and cache size. The compilers make architecture-specific decisions such as instruction selection, instruction scheduling, and vectorization, all of which can have a profound effect on the performance of applications.

Processor-specific code generation is controlled by the `-tp` option, described in the section '-tp <target> [,target...]' of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf. When an application is compiled without any `-tp` options, the compiler generates code for the type of processor on which the compiler is run.

14.2.1. Generating Generic x86-64 Code

To generate generic x86-64 code, use one of the following forms of the`-tp` option on your command line:

```
-tp px ! generate code for any x86-64 cpu type
```

```
-tp px-64 ! generate code for any x86-64 cpu type
```

Both of these examples are good choices for portable execution.

14.2.2. Generating Code for a Specific Processor

You can use the `-tp` option to request that the compiler generate code optimized for a specific processor. The PGI Release Notes contains a list of supported processors.

14.3. Generating One Executable for Multiple Types of Processors

PGI unified binaries provide a low-overhead method for a single program to run well on a number of hardware platforms.

All 64-bit PGI compilers for Windows can produce PGI Unified Binary programs that contain code streams fully optimized and supported for both AMD64 and Intel 64 processors using the `-tp` target option. You specify this option using PVF's Fortran | Target Processors property page. For more information on this property page, refer to the 'PVF Properties' section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

The compilers generate and combine multiple binary code streams into one executable, where each stream is optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

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

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

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

14.3.1. PGI Unified Binary Command-line Switches

The PGI Unified Binary command-line switch is an extension of the target processor switch, `-tp`, which may be applied to individual files during compilation using the PVF property pages described in the 'PVF Properties' section in the PGI Visual Fortran Reference Manual.

The target processor switch, `-tp`, accepts a comma-separated list of 64-bit targets and generates code optimized for each listed target.

The following example generates optimized code for three targets:

```
-tp k8-64,p7-64,core2-64
```

To use multiple `-tp` options within a PVF project, specify the comma-separated `-tp` list on both the Fortran | Command Line and the Linker | Command Line property pages, described in the 'PVF Properties' section in the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

A special target switch, `-tp x64`, is the same as `-tp k8-64, p7-64`.

14.3.2. PGI Unified Binary Directives and Pragmas

PGI Unified binary directives may be applied to functions, subroutines, or whole files. The directives and pragmas cause the compiler to generate PGI Unified Binary code optimized for one or more targets. No special command line options are needed for these pragmas and directives to take effect.

The syntax of the Fortran directive is:

```
pgi$[g|r| ] pgi tp [target]...
```

where the scope is g (global), r (routine) or blank. The default is r, routine.

For example, the following syntax indicates that the whole file, represented by g, should be optimized for both k8_64 and p7_64.

```
pgi$g pgi tp k8_64 p7_64
```

Chapter 15.
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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.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 Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

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

15.1. 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 using the Win64 convention.

The concepts in this section apply equally to using inter-language calling in PVF. While all of the examples given are shown as being compiled at the command line, they can also be used within PVF. The primary difference for you to note is this: Visual Studio projects are limited to a single language. To mix languages, create a multi-project solution.

Tip For inter-language examples that are specific to PVF, look in the directory:
`$(VSInstallDir)\PGI Visual Fortran\Samples\interlanguage\`

15.2. 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 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.
- 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.

15.3. 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 26, Fortran and C/C++ Data Type Compatibility, lists compatible types. If the call is to a Fortran subroutine, a Fortran `CHARACTER` function, or a Fortran `COMPLEX` function, call it from C/C++ as a function that returns void. The exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C/C++ as a function returning `int` whose value is the value of the integer expression specified in the alternate `RETURN` statement.

15.4. Upper and Lower Case Conventions, Underscores

By default on Linux, Win64, and macOS systems, all Fortran symbol names are converted to lower case. C and C++ are case sensitive, so upper-case function names stay upper-case. When you use inter-language calling, you can either name your C/C++ functions with lower-case names, or invoke the Fortran compiler command with the option `–Mupcase`, in which case it will not convert symbol names to lower-case.

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, and macOS systems, an underscore is appended to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C/C++ name space. Use these naming conventions:

- If you call a C/C++ function from Fortran, you should rename the C/C++ function by appending an underscore.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

15.5. Compatible Data Types

Table 26 shows compatible data types between Fortran and C/C++. Table 27, 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 26 Fortran and C/C++ Data Type Compatibility

Fortran Type (lower case)	C/C++ Type	Size (bytes)
character x	char x	1
character*n x	char x[n]	n
real x	float x	4
real*4 x	float x	4
real*8 x	double x	8
double precision	double x	8
integer x	int x	4
integer*1 x	signed char x	1
integer*2 x	short x	2
integer*4 x	int x	4

Fortran Type (lower case)	C/C++ Type	Size (bytes)
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 27 Fortran and C/C++ Representation of the `COMPLEX` Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8 8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

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

15.5.1. 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.

15.6. 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.

On the following systems, the compiler places the length argument(s) at the end of the parameter list, following the other formal arguments:

- On Linux and macOS systems
- On Win64 systems, except when using the option `-Miface=cref`

15.6.1. 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))
```

15.6.2. 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.

15.6.3. Complex Return Values

When a Fortran function returns a complex value, an argument needs to be added at the beginning of the C/C++ calling function's argument list; this argument is the address of the complex return value. COMPLEX Return Values illustrates the extra parameter, `cplx`, supplied by the caller.

COMPLEX Return Values

```
COMPLEX FUNCTION CF(C, I)
   INTEGER I
 . . .
END
```

```
extern void cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int i;
cf_(&c1, &i);
```

15.7. 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.

15.8. Examples

This section contains examples that illustrate inter-language calling.

15.8.1. 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 `f2c_main.exe` file should produce the following output:

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

15.8.2. 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 `c2fmain.exe` file should produce the following output:

```
TRUE v 11 -44 39.6 902 299
```

15.8.3. 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 `fmain.exe` file should produce the following output:

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

15.8.4. 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. When linking pgf77-compiled program units into C or C++ main programs, you need only link in -lpgftnrtl.

Chapter 16. PROGRAMMING CONSIDERATIONS FOR 64-BIT ENVIRONMENTS

You can use the PGI Fortran compilers on 64-bit Windows operating systems to create programs that use 64-bit memory addresses. However, there are limitations to how this capability can be applied. The object file format used on Windows limits the total cumulative size of code plus static data to 2GB. This limit includes the code and statically declared data in the program and in system and user object libraries. Dynamically allocated data objects can be larger than 2GB. This section describes the specifics of how to use the PGI compilers to make use of 64-bit memory addressing.

> The 64-bit PGI compilers are 64-bit applications which cannot run on anything but 64-bit CPUs running 64-bit Operating Systems.

This section describes how to use the following options related to 64-bit programming.

`-i8`	`-tp`

16.1. Data Types in the 64-Bit Environment

The size of some data types can be different in a 64-bit environment. This section describes the major differences. For detailed information, refer to the 'Fortran, C, and C++ Data Types' section of the PGI Visual Fortran Reference, www.pgroup.com/resources/docs/18.5/pdf/pvf18ref.pdf.

16.1.1. Fortran Data Types

In Fortran, the default size of the INTEGER type is 4 bytes. The `-i8` compiler option may be used to make the default size of all INTEGER data in the program 8 bytes.

16.2. Large Dynamically Allocated Data

Dynamically allocated data objects in programs compiled by the 64-bit PGI compilers can be larger than 2GB. No special compiler options are required to enable this functionality. The size of the allocation is only limited by the system.

16.3. Compiler Options for 64-bit Programming

The usual switches that apply to 64-bit programmers seeking to increase the data range of their applications are in the following table.

Table 28 64-bit Compiler Options

Option	Purpose	Considerations
-Mlargeaddressaware	[Win64 only] Generates code that allows for addresses greater than 2GB, using RIP-relative addressing.	Use -Mlargeaddressaware=no for a direct addressing mechanism that restricts the total addressable memory. This is not applicable if the object file is placed in a DLL. Further, if an object file is compiled with this option, it must also be used when linking.
-Mlarge_arrays	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Win64 does not support -Mlarge_arrays for static objects larger than 2GB.
-i8	All INTEGER functions, data, and constants not explicitly declared INTEGER*4 are assumed to be INTEGER*8.	Users should take care to explicitly declare INTEGER functions as INTEGER*4.

The following table summarizes the limits of these programming models under the specified conditions. The compiler options you use vary by processor.

Table 29 Effects of Options on Memory and Array Sizes

Condition	Addr. Math		Max Size Gbytes		
	A	I	AS	DS	TS
64-bit addr	64	32	2	2	2

Column Legend	
A	Address Type – size in bits of data used for address calculations, 64-bits.
I	Index Arithmetic -bit-size of data used to index into arrays and other aggregate data structures. If 32-bit, total range of any single data object is limited to 2GB.
AS	Maximum Array Size- the maximum size in gigabytes of any single data object.

DS	- max size in gigabytes combined of all data objects in .bss
TS	Maximum Total Size- max size in gigabytes, in aggregate, of all executable code and data objects in a running program.

16.4. Practical Limitations of Large Array Programming

The 64-bit addressing capability of the Linux86-64 and Win64 environments can cause unexpected issues when data sizes are enlarged significantly. The following table describes the most common occurrences of practical limitations of large array programming.

Table 30 64-Bit Limitations

array initialization	Initializing a large array with a data statement may result in very large assembly and object files, where a line of assembler source is required for each element in the initialized array. Compilation and linking can be very time consuming as well. To avoid this issue, consider initializing large arrays in a loop at runtime rather than in a data statement.

stack space	Stack space can be a problem for data that is stack-based. In Win64, stack space can be increased by using this link-time switch, where N is the desired stack size:`-Wl,-stack:N`
	`limit stacksize new_size ! in csh`
	`ulimit -s new_size ! in bash`

page swapping	If your executable is much larger than the physical size of memory, page swapping can cause it to run dramatically slower; it may even fail. This is not a compiler problem. Try smaller data sets to determine whether or not a problem is due to page thrashing.

16.5. Large Array and Small Memory Model in Fortran

The following example uses large, dynamically-allocated arrays. The code is divided into a main and subroutine so you could put the subroutine into a shared library. Dynamic allocation of large arrays saves space in the size of executable and saves time initializing data.

Large Array and Small Memory Model in Fortran

```
% cat mat_allo.f90
```

```
program mat_allo
  integer i, j
  integer size, m, n
  parameter (size=16000)
  parameter (m=size,n=size)
  double precision, allocatable::a(:,:),b(:,:),c(:,:)
  allocate(a(m,n), b(m,n), c(m,n))
  do i = 100, m, 1
```

```
    do j = 100, n, 1
      a(i,j) = 10000.0D0 * dble(i) + dble(j)
      b(i,j) = 20000.0D0 * dble(i) + dble(j)
    enddo
  enddo
  call mat_add(a,b,c,m,n)
  print *, "M =",m,",N =",n
  print *, "c(M,N) = ", c(m,n)
end
```

```
subroutine mat_add(a,b,c,m,n)
  integer m, n, i, j
  double precision a(m,n),b(m,n),c(m,n)
  do i = 1, m
    do j = 1, n
      c(i,j) = a(i,j) + b(i,j)
    enddo
  enddo
  return
end
```

```
% pgfortran -o mat_allo mat_allo.f90 -i8 -Mlarge_arrays -mp -fast
```

Chapter 17. CONTACT INFORMATION

You can contact PGI 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.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, Cluster Development Kit, PGC++, PGCC, PGDBG, PGF77, PGF90, PGF95, PGFORTRAN, PGHPF, PGI, PGI Accelerator, PGI CDK, PGI Server, PGI Unified Binary, PGI Visual Fortran, PGI Workstation, PGPROF, PGROUP, PVF, and The Portland Group are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright