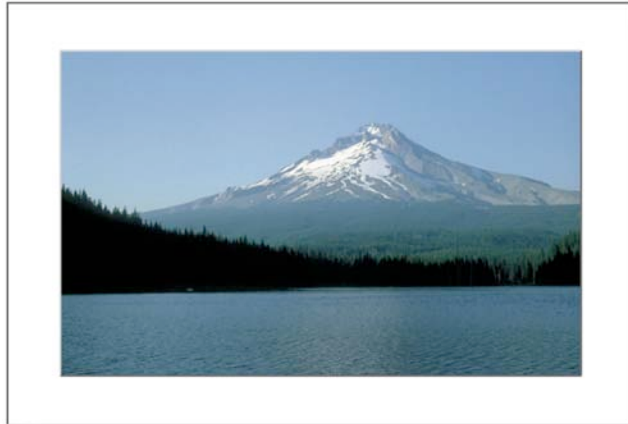
The Portland Group®



Migrating HPC Applications from UNIX to Windows

The Portland Group

Published: v2.0 September 2008

Contents

Introduction ... 1
Porting to Native Windows versus Windows SUA ... 2
Unix-like Development Environments for Native Windows ... 3
The Windows SUA Development Environment ... 5
The Microsoft Visual Studio Programming Environment ... 6
Operating System Specific Features ... 6

Common Migration Issues ... 8
File System and Pathname Differences—Native Windows ... 8
Drive Letters ... 8
Directory Separation Characters ... 9
Spaces in Paths and File Names ... 9
File System and Pathname Differences—SUA ... 9
Common Data Type Size Differences ... 10
Calling Conventions ... 10
Inter-language Calling ... 10

Case Studies ... 12
Overview ... 12
Case Study Conventions ... 12
Case Study Sample Code ... 12
Case Study 1: Build Environments ... 12
Native Windows Bash Shell ... 13
Native Windows Debugging ... 14
Native Windows — Visual Studio IDE ... 15
Subsystem for UNIX-Based Applications (SUA) ... 20
Case Study 2: Porting an MPI Program ... 22
Porting to Windows I : gethostname ... 22
Porting to Windows II : MPI ... 25
MSMPI Debugging ... 29
MPI in SUA ... 35
Case Study 3: Porting Shared Objects ... 36
Background ... 36
Starting Point: Unix ... 36
Porting Unix Shared Objects to Native Windows ... 37
Porting Unix Shared Objects to Windows SUA ... 41

Conclusion ... 42

References ... 43

Introduction

Migration of high-performance computing (HPC) applications from Unix-heritage operating systems, such as Solaris or Linux, to the Microsoft Windows operating system is potentially challenging for several reasons:

- Windows is primarily graphical user interface driven, while Unix development environments include extensive use of command-line driven shells, utilities and applications; Unix HPC applications typically include shell scripts and makefiles as an integral part of the compilation and execution environment, rather than the IDE-based projects and graphical deployments that are customary on Windows.
- Many small but significant issues affect experienced Unix developers porting to Windows as they work on their source code: line termination character differences, case sensitivity differences, treatment of the backslash character, etc.
- The file systems and filename conventions are different, which affects all lines of an application related to creating, reading, writing, and manipulating files; the Windows file system allows spaces in pathnames, whereas many Unix applications make assumptions that this will never occur.
- Use of shared object files is now pervasive on both Unix and Windows, but the implementations of shared object files on Unix (.so files) versus dynamically linked libraries on Windows (.dll files) involve differences that must be accommodated.
- Unix-heritage C, C++ and Fortran compilers support de facto standards regarding inter-language calling, data types, compiler options and directives that differ in various ways from the de facto standards on Windows.
- Operating system calls and traditional programming models, for example with respect to use of threads versus processes, are different on Unix versus Windows*.
- Access to some Windows services are best accomplished using Microsoft COM or .NET technologies. Both require additional learning to use effectively and they can introduce additional complications such as registration, etc.
- Software distribution and installation differ between Unix and Windows. While distributing a Windows application via a .zip file is certainly possible, most developers use special installation tools to handle the complexities of installing on Windows.

Fortunately, there are many tools and utilities from Microsoft and its partners that make Unix-to-Windows migration tractable and straightforward. Whether you need to migrate applications for use on Windows as part of an overall Windows migration strategy, or you simply wish to extend support for an HPC application into mixed Windows HPC Server® 2008 and Unix environments, this guide will explain the options you have and the steps you can take to achieve your Unix-to-Windows HPC migration goals.

* Third-party pthread libraries for Win32 exist (http://sourceware.org/pthreads-win32/) but there is no native support for pthreads on Win32.

Porting to Native Windows versus Windows SUA

When migrating professional-quality C, C++ or Fortran applications from Unix to Windows HPC Server 2008 you can migrate either straight to native Windows, or by using the Microsoft Subsystem for UNIX-based Applications (SUA).

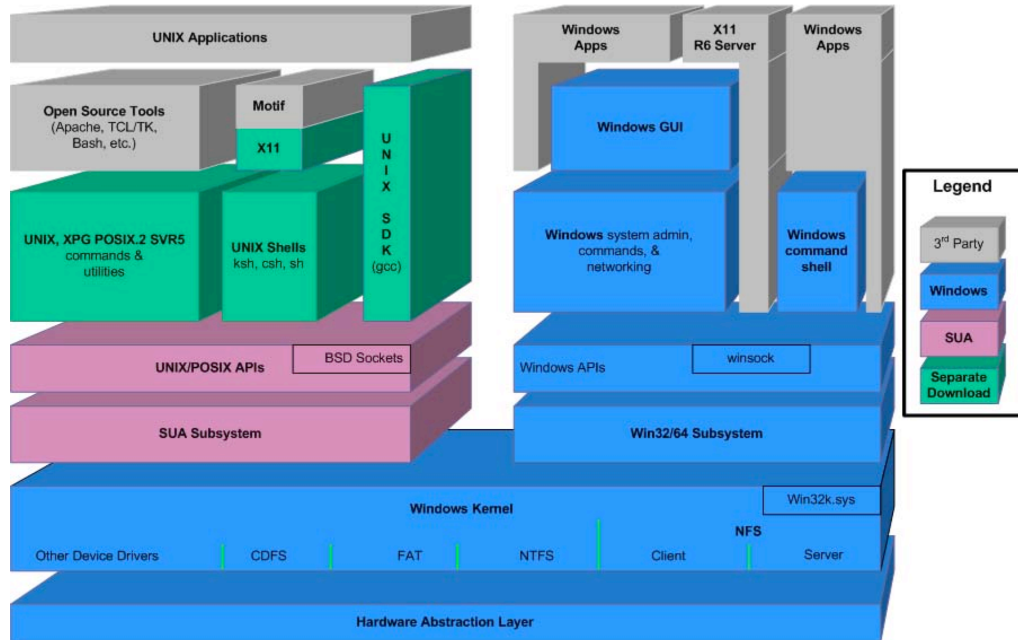


Figure 1 SUA Architecture†

SUA is a source-code compatible subsystem for compiling and running Unix-based applications on a computer running a Windows Server-class operating system. SUA is implemented alongside the Win32/x64 Subsystem as a second subsystem directly on top of the Windows kernel. Unix/POSIX APIs, shells, commands, utilities and software development tools are built and supported on top of this second subsystem, providing a software environment that delivers essentially all features and capabilities of a native Unix implementation running side-by-side with the traditional Windows software environment.

However, while it may seem logical to always start a migration project by leveraging the SUA environment, most HPC applications can be ported quite easily from Unix to native Windows using either a Unix-like command-level environment and compilers, or in some cases by migrating directly into a Microsoft Visual Studio 2008 project structure. There are several advantages to migrating directly to native Windows:

- Ability to run on Windows systems regardless of whether SUA is installed; SUA is available with most recent versions of Windows, but as an optional capability
- Access to MSMPI for porting of MPI-based cluster/parallel applications; MSMPI is not supported in the SUA environment
- Ability to use the Microsoft Visual Studio integrated development environment (IDE); while it's possible to attach to and debug SUA processes from within Visual Studio, SUA is not supported as a Visual Studio development target

† Copyright Microsoft Corp. Used with permission.

- Greater variety of tools and options for working with and managing your source code
- Full access to the Windows security model and user credentials

Two primary aspects affect the decision on whether you should migrate directly to native Windows: those related to your application development environment, and those related to source code portability across operating systems.

Unix-like Development Environments for Native Windows

A Unix HPC application development environment typically includes several discrete components:

- Command-line shells used to navigate the file system and manage files, such as *csh*, *sh*, *ksh* and *bash*
- File editors and manipulation utilities used to create, analyze and modify source code, such as *vi*, *emacs*, *sed*, *grep*, *diff*, *awk*, etc
- Scripting tools leveraged as part of source code processing or build environments such as perl, configure scripts, and shell scripts generally
- The *make* utility used to build binary executable files from source code
- Command-line driven optimizing Fortran, C and C++ compilers which support SIMD vectorization, automatic multi-core parallelization, inter-procedural optimization and optimized runtime and numerical libraries
- Command-line driven GNU Compiler Collection (GCC) compilers which support a large number of de facto standard C and C++ extensions and features used by Unix HPC applications

The ability to leverage a familiar development environment has a direct impact on your productivity—how quickly and easily you can begin the work required to get your HPC source code ported to Windows HPC Server 2008. Many HPC developers are extremely productive using traditional Unix software development tools, and would like to be able to continue to leverage these skills in a Windows environment. In addition, if applications are to be maintained across both Unix and Windows, it can be advantageous to retain a common development environment across both platforms. Most of the development environment tools traditionally used on Unix are also available on Windows, while the opposite is not true.

If you wish to continue working in a Unix-like development environment, that in itself is not a significant barrier to choosing a native Windows port. There are many available options that allow you to work in such an environment to create native Windows applications.

Commonly-used shell environment and command-level utilities packages include:

- The ***MKS Toolkit*** consists of multiple command shells, over 450 Unix-like command utilities, and custom utilities that simplify development across Unix and Windows operating systems—for more information, see http://www.mkssoftware.com/products/tk/ds_tkdev.asp
- ***Cygwin*** is a freely available toolkit including a Unix-like bash shell and utilities for working with source code, and the Cygwin DLL which provides an API including most Unix system calls. While the Cygwin development environment is extremely rich, linking an application against the Cygwin DLL requires that the resulting application be made available as open source—see http://www.cygwin.com/.

- ***PGI® Workstation*** from The Portland Group includes a Cygwin bash shell and utilities integrated with the PGI optimizing Fortran, C and C++ compilers, but structured as a tool chain that does not include or require the Cygwin DLL, allowing users the freedom to develop either proprietary or open source native Windows HPC applications—see http://www.pgroup.com/.

Commonly-used compiler packages include:

- ***PGI optimizing/parallel Fortran/C/C++ compilers*** for Intel and AMD x64 processors can be used in any of the shell environments listed above with functionality and look-and-feel nearly identical to their counterparts on Unix—see http://www.pgroup.com/.
- ***Intel optimizing Fortran/C/C++ compilers*** for Intel x64 processors provide functionality nearly identical to their counterparts on Unix with a command interface and options compatible with the traditional Windows DOS-style shell command environment—see http://developer.intel.com/.
- ***Microsoft Visual C++ compilers*** are accessible from the DOS command prompt. DOS also includes the Microsoft Program Maintenance Utility (nmake.exe), a Unix-like tool for building projects from commands stored in a description file. Current versions of Microsoft compilers do not automatically vectorize.
- ***Cygwin GCC compilers*** include most of the GNU compiler collection compilers and gdb debugger, but link by default against the Cygwin DLL (see above).

Figure 2 shows an example screen shot of a typical Windows desktop running the PGI Workstation compilers and development tools side-by-side with the Microsoft Visual Studio IDE. With access to a bash command shell to navigate the file system and source code directory trees, a vi editor to create and modify source files (emacs for Windows is available through various vendors), a modern Unix-compatible *make* utility, Unix-compatible OpenMP parallel Fortran, C and C++ compilers, and graphical MPI/OpenMP parallel debugging and profiling tools, this desktop includes most of the features you would expect in a typical Unix cluster HPC development environment.

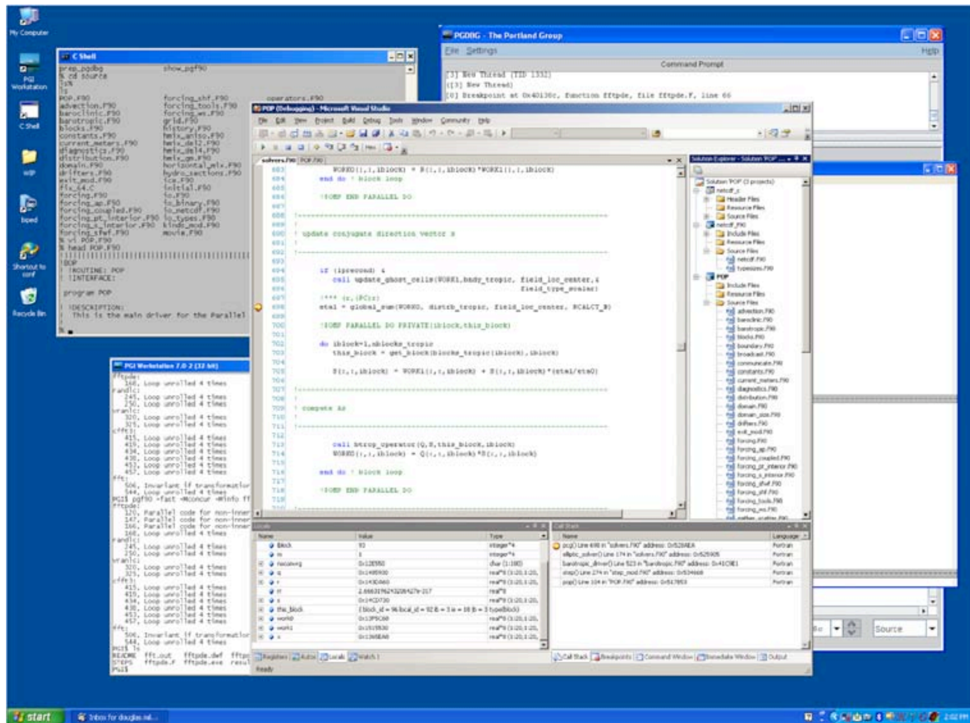


Figure 2 Unix-like shells, compilers and tools for Windows running side-by-side with Microsoft Visual Studio

All of the compilers and tools in the PGI Workstation package, however, are structured to create native Windows executables using the Microsoft tool chain and header files, and are interoperable with MSMPI to enable creation of native MSMPI applications for Windows HPC Server 2008 clusters. As you will see in the sections which follow, you can use PGI Workstation or any of the other packages listed above to port or create native Windows HPC applications in a development environment that leverages existing Unix program development skills and expertise.

The Windows SUA Development Environment

The Windows SUA application development environment includes a default version of the GNU Compiler Collection (GCC) Unix SDK compilers and tools (32-bit only) built on top of the Unix/POSIX APIs supported by the SUA subsystem.

In addition, there are packages available that provide many additional commonly-used or more current shells, utilities and compilers:

- *Interop Systems Developer Bundle and Tools Warehouse* includes additional SUA command shells (including *bash*), GCC 4.2 compilers and tools, the emacs editor and numerous other tools and utilities—see http://www.interopsystems.com/.
- The *PGI Unix to Windows Migration Package* supports development of OpenMP parallel Fortran, C and C++ HPC applications within the SUA environment. Contact PGI Sales for more information at sales@pgroup.com.

Because these compilers and tools are built within the SUA environment and create applications native to the SUA subsystem, they provide a development environment with maximum support for Unix compatibility rather than simply a Unix-like development environment for native Windows. As outlined in previous sections, there are limitations

inherent in an SUA port including the lack of interoperability with MSMPI. However, the SUA environment does provide a level of Unix source code compatibility that can minimize the effort required to port certain types of HPC applications to Windows.

The Microsoft Visual Studio Programming Environment

An advantage of working on Windows is access to one of the most powerful integrated development environments available. Microsoft Visual Studio provides the ability to develop, build, run and debug applications without leaving the IDE. The Visual Studio solution explorer provides a graphical representation of a program's structure, letting you see how things are laid out at a glance. This visual representation can help engineers quickly understand the organization of an application, and it may aid in future design decisions.

Fortran language packages for Visual Studio, which include PGI Visual Fortran® and Intel Visual Fortran, are widely available and interoperable with Visual C++. Syntax colorization in the editor speeds up programming, especially for coders who may not be language experts yet. Migrating Fortran-only programs to Visual Studio can be as simple as adding source files to a new solution and selecting Build Solution from the Build menu; this feature works for large and small applications alike. Migrating mixed-language programs may require additional upfront design considerations because Visual Studio separates projects by language—every project yields a single image, be it library or executable. Even with a mixed-language application, though, actual set-up time in the IDE is minimal once you are comfortable with your design.

Highlights of Visual Studio and PGI Visual Fortran features include:

- Application development is organized by platform (i.e., Win32, x64) and configuration (i.e., Debug, Release), any combination of which can be built from within the IDE or at the command line.
- Build dependency information, including dependencies introduced by Fortran modules, is generated automatically.
- Language support includes syntax colorization, statement completion, and Fortran intrinsic method tips.
- Errors in compilation output act as links to the source location causing the build failure.
- Seamless debugging in mixed-language environments includes display of local variables and variable rollovers, call stack, source- and assembly-level stepping, and register access.

Whether or not you are familiar with IDEs in a Unix environment, it is worth exploring Visual Studio as you prepare to migrate your applications to Windows.

Operating System Specific Features

The portability of your source code itself depends on many factors, including your coding style and the extent to which you have employed Unix-specific features. Regardless of the tool or technique used, in most cases, equivalent functionality exists on native Windows. The amount of effort required to port source code to use Windows functionality can vary widely depending on the application. For example, many HPC applications contain few, if any, system calls and can be re-built on native Windows with no modifications to the source code.

Examples of build script and source code characteristics that are relatively straightforward to modify in porting an HPC application to native Windows include:

- Change file directory names from single-rooted (/usr, /local, /tmp) Unix-style to Windows-style directory names (C:\, E:\).
- Modify file processing scripts and code to handle directory and file names that include space characters and to ensure that no filenames include the colon (':') character, which is significant in Windows file names.
- Modify file processing scripts and code to account for the fact that the directory separator on Windows ('\') is the escape character on Unix.
- Change filenames to account for lack of case sensitivity on Windows; for example, Fortran files with a .F extension for pre-processing should be changed to a .F90 extension to avoid unintended file overwrites (file.F and file.f are the same file on Windows).
- If necessary, modify long data types, which are 8 bytes on Unix versus 4 bytes on Windows, to long long which is 8 bytes on both platforms (64-bit only).
- If necessary, modify compiler invocations in makefiles and build scripts to include the –o $@.o option to produce .o files rather than the default .obj files; this can minimize other required changes to the makefile or script.

Conversely, below are examples of build script and source code characteristics that are more difficult to accommodate in a port to native Windows, but which can be accommodated easily by porting under Windows SUA:

- Use of Unix-specific system calls (e.g. fork())
- Extensive use of Unix-style threading and/or use of thread condition codes
- Applications which rely extensively on Unix-specific process hierarchies
- Applications with X Windows graphical user interface components

If you want to get applications such as these up and running quickly on Windows, or if they will have a limited use or lifetime, it may be advisable to do an initial port in the SUA environment.

Fortunately many HPC applications, and in particular Fortran HPC applications, make relatively few assumptions about and have few dependencies on the underlying operating system. It is not uncommon to be able to port such applications from Unix to Windows HPC Server 2008 with very few changes, and in many cases the bulk of the changes are in the build scripts and environment rather than in the source code itself. In the sections that follow, we will use case studies and small tractable examples to illustrate how to overcome the most common porting issues you are likely to encounter.

Common Migration Issues

The objective of this white paper is to convey the flavor of the porting process, along with some very explicit examples of required changes and how they can be addressed. Every application port involves its own unique challenges. While it's impossible to cover every possibility exhaustively, we discuss below the issues that are most commonly encountered and that can be easily addressed with appropriate guidance and preparation.

This part of the paper opens with several general discussion sections covering these common migration topics:

- File System and Pathname Differences—Native Windows
- File System and Pathname Differences—SUA
- Common Data Type Size Differences
- Calling Conventions
- Inter-language Calling

Then, three case studies will take an in-depth look at these porting topics:

- Build Environments
- Unix MPI to MSMPI
- Shared Objects

Let's start with the general discussion topics.

File System and Pathname Differences—Native Windows

Almost anyone who has spent time working on both Unix and Windows systems already knows that the file system and pathnames between these two operating systems are significantly different. These differences are important when planning a source code port from Unix to Windows, and it is useful to be aware of what issues to watch out for.

Here is a common native Windows pathname:

```
C:\Program Files\Microsoft Visual Studio 9.0
```

This path illustrates three major differences between Unix and Windows paths: the use of drive letters, the difference in directory separation character, and the prevalence of spaces. We expand on each of these issues below.

Drive Letters

The Windows OS labels drives (local, media, mapped/mounted/networked) using letters of the alphabet. Most full pathnames in the native Windows environment begin with a drive letter and a colon. The main system drive is commonly the 'C:' drive, but this is not guaranteed. The %SYSTEMDRIVE% system variable is set to the host's system drive. To check this on a given system, you might run:

```
DOS> echo %SYSTEMDRIVE%
E:
```

You can use this variable in scripts and batch files to improve their portability between different Windows systems.

Directory Separation Characters

The native Windows directory separation character is the backslash ('\'). In the Unix environment, the forward slash ('/') is used to separate directories. Further, on Unix the backslash acts as the escape character. Source code is not the only place where these differences can cause difficulty during a port. Scripts, config files, makefiles and their like often contain paths. Passing backslashes between these types of files often requires "double-backslashing" to get the correct path passed through to its final destination.

In most cases, you will only need to use a single extra backslash (i.e., '\\' instead of just '\'). Once in a while you might run into a more pathological case. The worst-case scenarios typically involve multiple tools and layers of shell scripts and makefiles. Someday you may need to use something like the following code, which was taken from a batch file-invoked perl script running pattern matching on a registry file:

```
my $tmpdir = "$ENV{SYSTEMDRIVE}\\\\\\\\tmp\\\\\\\\";
```

But don't worry—most paths will not require this level of escaping. Keep the issue in mind, though, and try escaping backslashes when your paths are not being resolved correctly.

Spaces in Paths and File Names

Windows path and file names can (and often do) contain spaces. While many modern Unix systems allow spaces in directory and file names, spaces have not historically been used by Unix users. It is common for applications developed on Unix platforms to include assumptions about what spaces mean. For example, Unix-developed code may tacitly assume that spaces are separators.

File System and Pathname Differences—SUA

Although more similar to Unix than native Windows, the SUA file system is different from both. We will start our examination using the following familiar Windows path:

```
C:\Program Files\Microsoft Visual Studio 9.0
```

The SUA representation of the above path follows:

```
/dev/fs/C/Program Files/Microsoft Visual Studio 9.0
```

The directory separator is, as on Unix, the forward slash. Spaces are allowed in paths but when such paths are used, they must either be quoted:

```
%  cd "/dev/fs/C/Program Files/Microsoft Visual Studio 9.0"
```

Or the spaces must be escaped with backslashes:

```
% cd /dev/fs/C/Program\ Files/Microsoft\ Visual\ Studio\ 9.0/
```

The obvious difference between SUA and Unix is the use of the `/dev/fs/<drive>` prefix. This prefix is used as the base of all SUA paths that are outside of the SUA directory tree.

Paths to anything located within the SUA directory tree, typically located in `C:\WINDOWS\SUA`, do not include the prefix. In fact, the entire first part of the path

(`/dev/fs/<drive>/WINDOWS/SUA`) is omitted. For example, the following Windows path to SUA's tmp directory:

```
C:\WINDOWS\SUA\tmp
```

Is represented as the following path under SUA:

```
/tmp
```

Common Data Type Size Differences

Any paper on Unix to Windows migration would probably not be complete without some mention of the size difference of common data types between operating systems and platforms (32 or 64 bit). The following table summarizes these differences for three C/C++ data types. In particular, note the difference between the size of a `long` on 64-bit Unix systems and 64-bit Windows.

Operating System	int	long	pointer
Windows 32	4	4	4
Linux 32	4	4	4
SUA 32	4	4	4
Windows 64	4	4	8
Linux 64	4	8	8
SUA 64	4	8	8

Table 1: Size in Bytes of Three C/C++ Data Types

Calling Conventions

The Fortran and C calling conventions for Unix are similar to those used on Windows x64 and SUA. The Fortran calling convention for Win32, however, can vary significantly from that used by Unix and Windows x64. Win32 calling conventions and their implementation can differ by compiler; refer to the documentation for your Fortran, C, and C++ compilers for calling convention details.

At the assembly code level, calling conventions differ significantly between Windows and Unix because the Application Binary Interface (ABI) for Windows differs from the ABI for Unix. For example, the set of registers used to pass arguments to functions on 64-bit Windows differs from the register set used to pass arguments to functions on 64-bit Unix.

Inter-language Calling

Inter-language calling, typically between Fortran and C/C++ in an HPC application, can be another source of frustration in a port. Different compiler vendors have developed different ways to solve the compilation issues involved in inter-language calling. Refer to the documentation for the compilers you are using in your port to learn what source modifications you need to make to enable inter-language calling. As an example of the types of source changes that might be required, review the following guidelines for porting mixed Fortran and C/C++ code when using the PGI compilers:

- C++ functions should be declared `extern "C"`, or the C++ code should be compiled as C code.

- C functions to be called by Fortran must have an underscore ('_') appended to their names. If Fortran calls `mysub`, declare the C function as `mysub_`.
- If you are mixing Fortran and C/C++ in a Visual Studio solution, you will notice that each Visual Studio project is limited to a single language. If you have a Fortran main project, then you would set up a Visual C++ library project for the C/C++ subroutines.

Case Studies

Overview

Three case studies are presented in the sections that follow:

- Build Environments: Explore the available build environments while building a single Fortran source file for both native Windows and SUA. If you are already familiar with working on Windows, you may be able to skip this section.
- Porting MPI: Port a simple MPI program written in C for a Unix-like OS to Windows.
- Shared Objects: Address the conceptual differences between shared objects on Unix and Windows, and walk through the process of porting this type of library.

Case Study Conventions

The case studies follow a few conventions in order to simplify the examples:

- Error checking, an important part of any software program, has not been included in coding examples in order to focus attention on the topics being illustrated.
- Microsoft Visual Studio solutions have been built with Microsoft Visual Studio 2008 and PGI Visual Fortran.
- The x64 compilers and tools from The Portland Group and the Microsoft C/C++ compiler are used throughout the case studies.
- The value of %SYSTEMDRIVE% is assumed to be 'C:'.

Make adjustments to the sample commands and coding examples as needed to accommodate differences in your environment.

Case Study Sample Code

A .zip file archive containing all the source code, scripts, makefiles, solution and project files used in the case studies is available from The Portland Group website at:

http://www.pgroup.com/lit/pgi_source_sample_unix2win.zip

The following structure is used in the Case Study directories:

unix2win\case1	Case study 1
unix2win\case2	Case study 2
unix2win\case3	Case study 3

Each case study's text contains everything needed to reproduce the test case.

Case Study 1: Build Environments

This case study addresses the following application development steps:

- Set the environment
- Build
- Run

- Debug

Across these Windows development environments:

- Native, command-line
- Native, Visual Studio IDE
- SUA, command-line

We will use a simple Fortran main program that reads program arguments from the command line and prints them out. This code will build without modification on Unix, native Windows and SUA. Without source code porting issues, this example will allow the exploration of the build environment differences between platforms.

Here is the source code used for Case Study 1 (unix2win\case1\src\prog.f90):

```
program printargs

    integer i, argc, iargc
    character(500) arg

    argc = iargc()

    do i = 0, argc
        call getarg(i, arg)
        print 10, i, arg
    end do

10 format ( I4.4, ' : ', A200 )

end
```

Native Windows Bash Shell

PGI Workstation for Windows includes a bash shell environment from Cygwin. This bash shell provides a Unix-like command-prompt and a set of command-level tools including a version of *make* compatible with most modern makefiles. The bash environment includes a slew of other tools found on Unix, including *vi, grep, awk, sed* and *find*.

The PGI Workstation icon on your desktop invokes a bash shell with the environment pre-configured to run the PGI compilers and tools. Double-click it to open this bash shell:

> ***Helpful Hint #1***
>
> Windows shells have Mark and Paste commands for copying text out of and bringing text into a window. These tasks have keyboard shortcuts:
>
> Mark: Alt + Space, E, K
>
> Copy: Mark + <ENTER>
>
> Paste: Alt + Space, E, P
>
> In Quick Edit Mode, Marking is always enabled; set this mode in the Option tab of the window's Properties dialog.

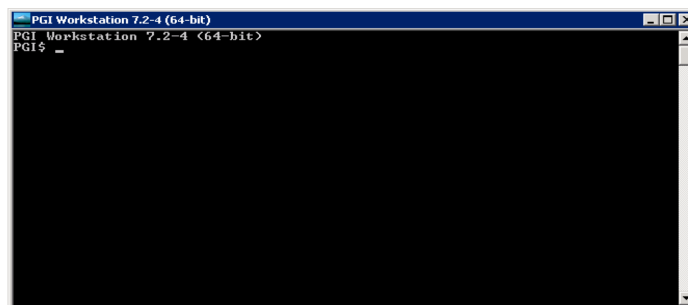


Figure 3: PGI Workstation bash shell

Build the application

As long as the environment is set to pick up the compilers you want to use, it is easy to build the example code and run from the command line:

```
PGI$ cd unix2win/case1
PGI$ pgf95 –g src/prog.f90 -o prog

PGI$ prog
0000 : c:\tmp\unix2win\case1\prog.exe
```

The Cygwin bash environment includes *make* so you can use a makefile, just like on Unix.

```
PGI$ cd bash
PGI$ cat makefile

prog.exe: ../src/prog.f90
    pgf95 -g ../src/prog.f90 -o prog

clean:
      @rm -f prog.exe *.dwf *.pdb

PGI$ make clean
PGI$ make prog.exe
pgf95 -g prog.f90 -o prog
```

Native Windows Debugging

PGI Workstation includes a native Windows debugger called PGDBG®. This debugger runs in both command line mode and in GUI mode, and can be invoked from either a command prompt or a bash shell.

To invoke the command-line debugger on prog.exe, which we have compiled with -g to produce debug information, provide the -text option to PGDBG as follows:

> ***Helpful Hint #2***
>
> To find out what commands are available in the debugger, use the help command by itself:
>
> pgdbg> help
>
> To find out more about any one command, provide that command as a help argument:
>
> pgdbg> help break

```
PGI$ pgdbg -text prog
PGDBG 7.2-4 x86-64 (Cluster, 256 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc. All Rights Reserved.
Loaded: c:\tmp\unix2win\case1\bash\prog.exe
```

If you're familiar with PGDBG from working with it on Unix, then you'll notice that the interface and commands are all the same.

Once the program is loaded, set a breakpoint and run to it:

```
pgdbg> break main
(1)breakpoint set at: printargs line: "../src/prog.f90"@6 address: 0x140001134
1
pgdbg> run
Loaded: C:/WINDOWS/system32/ntdll.dll
Loaded: C:/WINDOWS/system32/kernel32.dll
Breakpoint at 0x140001134, function printargs, file ../src/prog.f90, line 6
 #6:         argc = iargc()

pgdbg>
```

Use the `cont` command to run (continue) the program to completion:

```
pgdbg> cont
(Process Exited)
```

To exit the debugger, use quit:

```
pgdbg> quit
```

This case study used PGDBG in text mode. By omitting the -text option on start-up, the PGDBG GUI will be invoked. Use of this GUI is discussed in detail in the second case study.

Native Windows — Visual Studio IDE

You can use the Microsoft Visual Studio IDE to build and debug native Windows applications. PGI Visual Fortran (PVF®) represents the integration of the PGI Fortran compilers with Visual Studio.

This section provides an introduction to PGI Visual Fortran as well as basic information about how things work in Visual Studio. It contains a step-by-step example of how to create a PVF project and add the sample code—prog.f90—introduced in this case study. Once the project is created, it is simple to build, run and debug the program. The complete example, including solution and project files, is provided alongside this tutorial.

We will start the discussion by defining some terms commonly used in Visual Studio (VS). If you're already familiar with PVF or are comfortable with project creation in VS, this section might be review for you. If that's the case, you may want to skip ahead to the next section.

Two terms used frequently when working in the Visual Studio IDE are *solution* and *project*. For consistency of terminology, it is useful to discuss these at the outset:

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 Microsoft Visual C^{++} or PGI Visual Fortran, but you can have projects of different languages in the same solution.

Create the Project

We want to start by opening Visual Studio and creating a project to which we can add prog.f90.

Open PGI Visual Fortran from the Start menu. Creating a new project and its solution is easy. Select New from Visual Studio's File menu, then select "Project…" The New Project dialog box will appear.

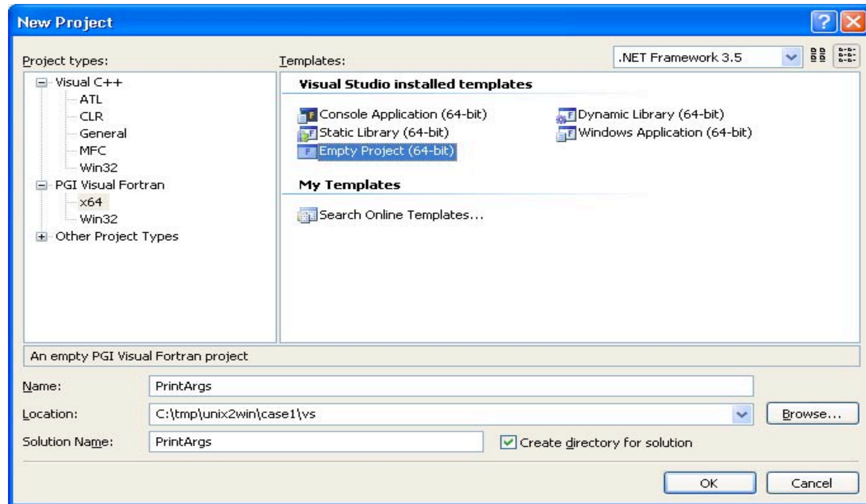


Figure 4: PVF New Project Dialog

To create a new 64-bit empty project, follow these steps:

1. In the Project types pane, select PGI Visual Fortran and x64.
2. In the Templates pane, select Empty Project (64-bit). Note that while both 32- and 64-bit projects can be created on 64-bit systems, only 32-bit projects can be created on 32-bit systems.
3. In the Name field located at the bottom of the dialog box, type: PrintArgs. Click OK.

After the solution is created, you should see the Solution Explorer window in PVF:

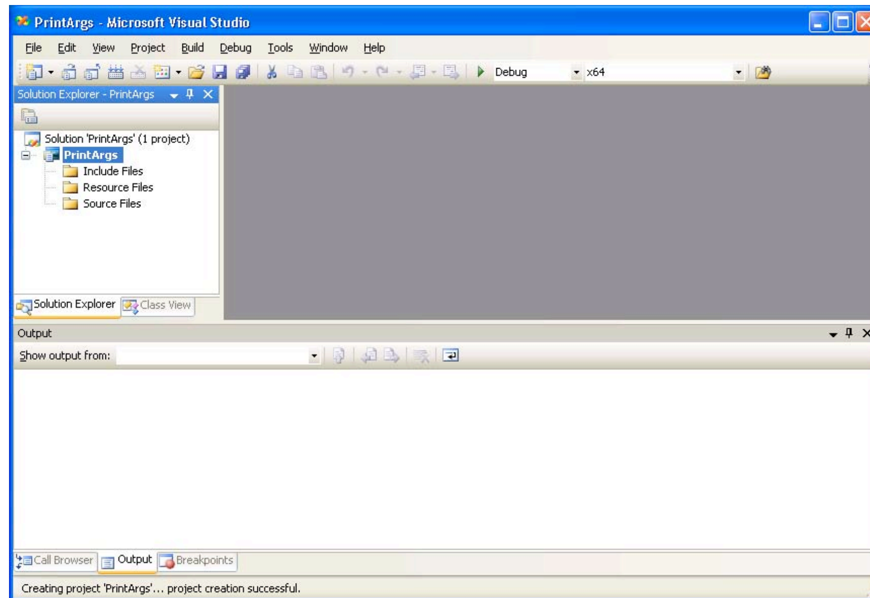


Figure 5: PrintArgs Solution in VS Solution Explorer

If the solution explorer is not open by default, open it from the View menu.

Add the Source File

Adding an existing source file to a VS project is a trivial process. We'll add prog.f90 to our project using the following steps:

1. In the Solution Explorer, right-click on the project name (i.e., PrintArgs).
2. From the context menu, select Add | Existing Item ...

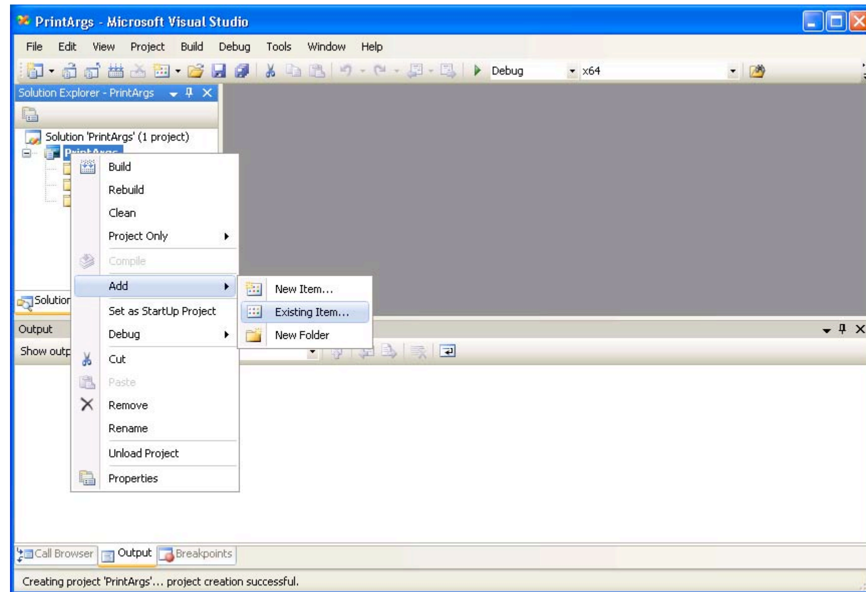


Figure 6: PVF Add Item Menu

3. In the Add Existing Item dialog box, browse to the location of prog.f90. We will use the version of this file provided in unix2win\case1\src.

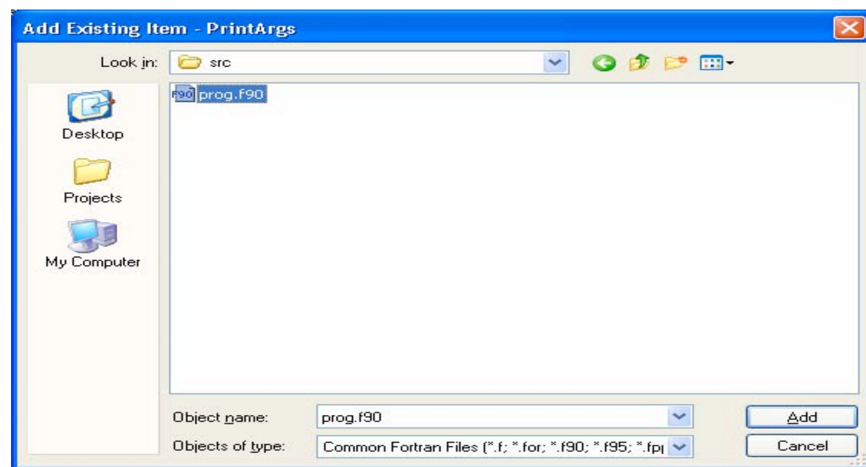


Figure 7: PVF Add Item Dialog

4. Select prog.f90, then click Add.

Build the Project

Now that the source file is in the project, the project is ready to be built. Select Build Solution from the Build menu. The Output window shows the results of the build.

> ***Helpful Hint #3***
>
> The Add Existing Item dialog will allow you to select multiple files at one time using Ctrl+Click or Shift+Click. This feature lets you create a project with a large number of source files very quickly.

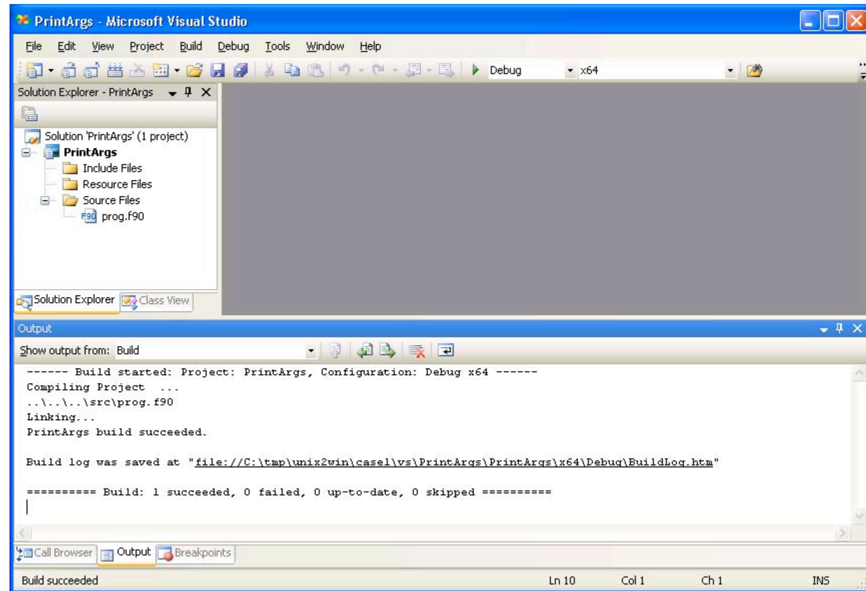


Figure 8: PrintArgs build output

Run the Program

You can run the program you just built from within the IDE. Select Start Without Debugging from the Debug menu. This action launches a command window in which you see the output of the program. It looks similar to this:

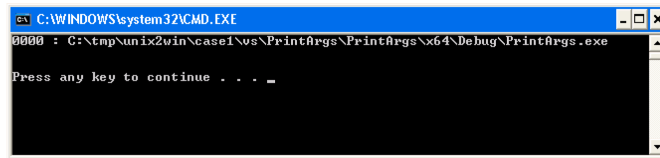


Figure 9: PrintArgs runtime output

To exit the command window, press any key.

Property Pages

Almost all of the information that you would put in a Unix makefile should be put into what Visual Studio calls property pages. If you've never worked with Visual Studio before, take a minute to open your project's property pages. Right-click on the PrintArgs project and select Properties from the context menu.

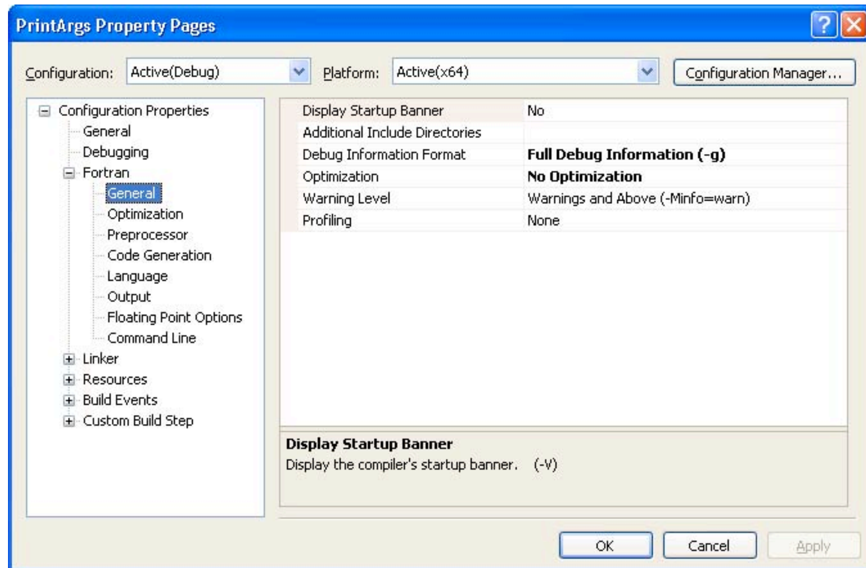


Figure 10: A PVF Property Page

Debug the Program

It is as easy to debug the program from within the VS IDE as it is to run it. The following steps will walk you through setting a breakpoint in the source and starting up the debugger.

1. Open prog.f90 in the editor by double-clicking on the filename in the solution explorer.
2. Left-click in the far left side of the editor on the line where you want to put the breakpoint. A red circle appears to indicate that the breakpoint is set.

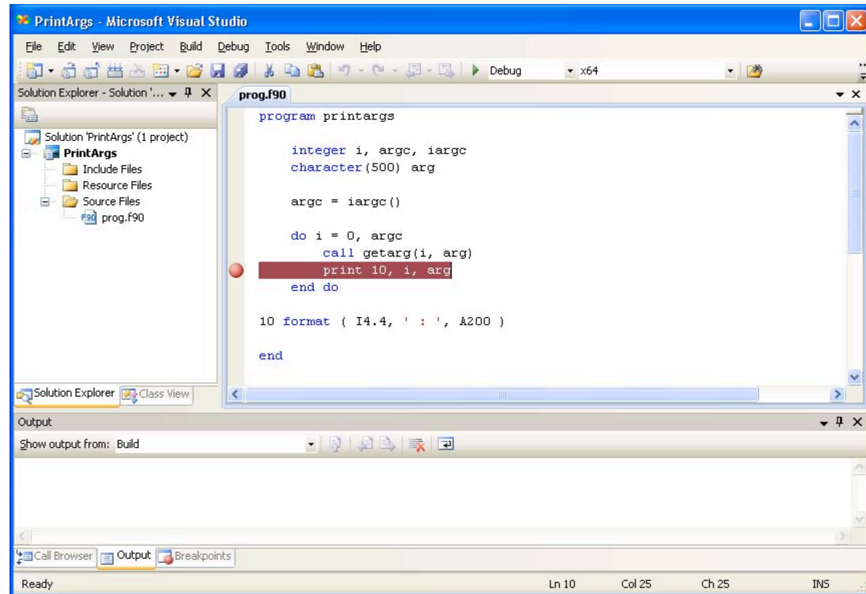


Figure 11: Setting a breakpoint in PrintArgs

3. Select Start Debugging from the Debug menu to start the PGI Visual Fortran debug engine. If you have made any changes to the project since the last time you built it, you will be prompted to rebuild before debugging. Once debugging begins, the debug engine stops execution at the breakpoint.

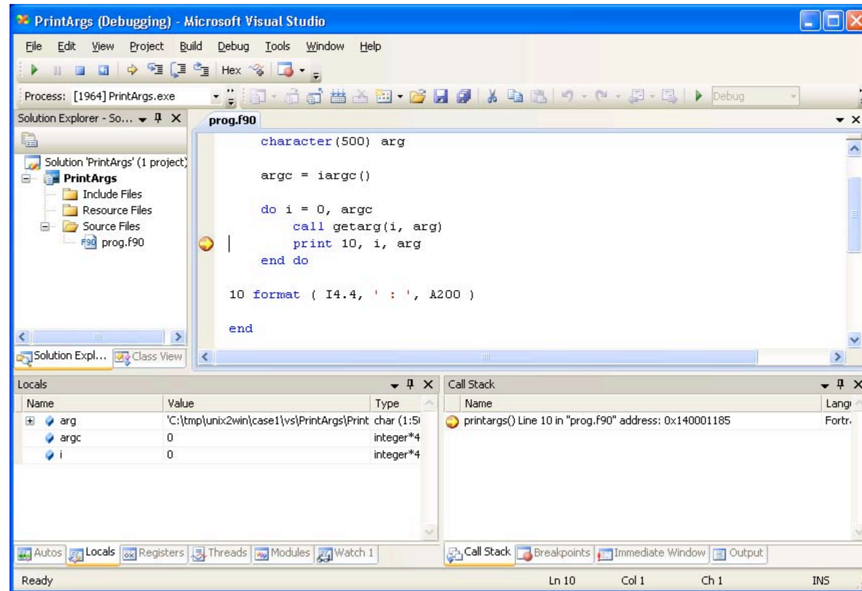


Figure 12: Stopped at a breakpoint in PrintArgs

4. Use the Debug menu's Step Over command to step over the breakpoint.
5. Proceed with execution using Continue. The program will exit.

Now that you've had some experience with Visual Studio, you should begin thinking about how you might go about porting your Unix applications into a VS solution in order to take advantage of the features of this IDE.

Subsystem for UNIX-Based Applications (SUA)

We will now switch back to working at the command line as we examine the process of building, running and debugging within the SUA environment. When the Utilities and SDK for UNIX-based Applications has been installed on a system where the SUA component of the OS has been enabled, you will have access to Korn and C shells containing a wide variety of tools typically found on Unix including *vi*, *grep*, *awk*, *sed* and *find*.

You can open C and Korn shells from the Start menu. These links are located under the Subsystem for UNIX-based Applications link.

The example code in this section will use C shell syntax. If you would rather use the Korn shell (or a different, third-party shell), adjust the syntax as needed for your shell environment.

Set the environment

Just like on a Unix system, you will need to set up your SUA environment before you can build anything. Setting up the environment to access the PGI Workstation for SUA Fortran, C and C++ compilers involves a few setenv commands:

```
% setenv PGI /opt/pgi
% setenv PATH "$PGI/sua64/7.2-4/bin:$PATH"
% setenv PGIBIN "$PGI/sua64/7.2-4/bin"
```

You might also want to add the current working directory to PATH. This step is not necessary to run the compilers but makes running the resulting applications easier.

```
% setenv PATH ".:$PATH"
```

If you are going to do a lot of work within SUA, you'll want to add these and other environment setup commands to a .csh file that can be added to your shell's start-up or sourced from the command line. Here is an example:

```
% cd unix2win/case1/sua
% cat setenv.csh

# source this file

setenv PGI /opt/pgi
setenv PATH ".:$PGI/sua64/7.2-4/bin:$PATH"
setenv PGIBIN "$PGI/sua64/7.2-4/bin"

% source setenv.csh
```

Check to see that the environment is set correctly by invoking the PGI Fortran compiler:

```
% pgf95 -V

pgf95 7.2-4 64-bit target on 64-bit SUA
Copyright 1989-2000, The Portland Group, Inc.  All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc.  All Rights Reserved.
```

Build the application

Once you've got the environment set up correctly, you can invoke the compiler from the command line to compile a source file. Here we compile the same prog.f90 file that we have used throughout this case study:

```
% pgf95 -g ../src/prog.f90 -o prog
```

Run the application

As soon as the compiler finishes building the executable, you can run it from the command line:

```
% prog
0000 : prog
```

Use make to Build

The SUA environment includes the *make* utility. You can use this version of *make* with many of the makefiles developed in the Unix environment.

The following example shows how you might use a makefile to compile prog.f90:

```
% cat makefile
prog: ../src/prog.f90
    pgf95 -g ../src/prog.f90 -o prog

clean:
      @rm *.dwf prog

% make clean
% make prog
pgf95 -g ../src/prog.f90 -o prog
```

Case Study 2: Porting an MPI Program

This case study shows you how to take an MPI program written in C for the Unix platform, port it to run on Windows using MSMPI, and run it under a debugger.

The case study will also cover some common development environment issues you may encounter when beginning to work on a Windows cluster.

Starting Point: Unix

We'll start our port to Windows MPI with a simple MPI program, written in C (unix2win\case2\unix.c):

```
#include <unistd.h>
#include "mpi.h"

#define BUFFLEN 32

void main(int argc, char **argv)
{
    char hname[BUFFLEN];

    MPI_Init(&argc, &argv);

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);

    MPI_Finalize();
}
```

As a starting point, let's compile this file on a Unix system. We will use the PGI C compiler for Linux and the MPICH1 libraries:

```
% pgcc -Mmpi=mpich1 unix.c -o unix
```

Now we will run the binary on a four-node Unix cluster, with the names of the nodes in the file "host.list". The output of this run will serve as our baseline so we will know when we've succeeded in our port to Windows.

```
% /opt/mpi/mpich/bin/mpirun -np 4 -machinefile host.list unix
My name is node1
My name is node2
My name is node3
My name is node4
```

Porting to Windows I: gethostname

If you examine the source code we are attempting to port, you will notice that the code does two things: it uses gethostname, and it uses MPI. To make this port easier to follow, we will tackle these two parts of the code separately. We will port the use of gethostname first, and the use of MPI second.

The MPI calls have been removed in the following version of the source code (unix2win\case2\win1.c):

```
#include <unistd.h>

#define BUFFLEN 32

void main(int argc, char **argv)
{
    char hname[BUFFLEN];

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);
}
```

For this case study, we will use Microsoft's *C/C++* compiler, *cl*. For simplicity's sake, we are going to use *cl* from the command line for this part of the case study. A quick way to get a command line environment configured for using *cl* is to open the Visual Studio 2008 x64 Win64 Command Prompt. Look for this entry on the Start Menu under Microsoft Visual Studio 2008 | Visual Studio Tools. A similar option is available for Visual Studio 2005.

Will building the source on Windows work without modification? Let's try it!

```
C:\tmp\unix2win\case2> cl win1.c /Fewin1
Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win1.c
win1.c(1) : fatal error C1083: Cannot open include file: 'unistd.h': No such
file or directory
```

We'll remove the reference to "unistd.h" (unix2win\case2\win2.c):

```
#define BUFFLEN 32

void main(int argc, char **argv)
{
    char hname[BUFFLEN];

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);
}
```

And try again:

```
C:\tmp\unix2win\case2>cl win2.c /Fewin2
Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win2.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:win2.exe
win2.obj
win2.obj : error LNK2019: unresolved external symbol gethostname referenced
in function main
win2.exe : fatal error LNK1120: 1 unresolved externals
```

Either gethostname is not available on the Windows platform or we need to link it in from some non-default library. One way to find out what to do is to search the MSDN documentation, either online or as installed on your system. Every MSDN Library page referring to a routine or constant includes a "Requirements" section that describes which headers and libraries are needed to access the feature. Looking for gethostname in MSDN

reveals that a version of this routine is available by including Winsock2.h. We'll add this include file to our code (unix2win\case2\win3.c):

```
#include <Winsock2.h>

#define BUFFLEN 32

void main(int argc, char **argv)
{
    char hname[BUFFLEN];

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);
}
```

We will also add Ws2_32.lib to the link line.

```
C:\tmp\unix2win\case2>cl win3.c /Fewin3 /link /defaultlib:Ws2_32.lib
Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win3.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:win3.exe
/defaultlib:Ws2_32.lib
win3.obj
```

That worked. Let's run the file:

```
C:\tmp\unix2win\case2>win3
My name is ó*
```

Well, it runs. But the output is not exactly what we expected so we need to debug it.

Referring back to the gethostname documentation indicates possible error codes that can be returned by this function. The description of one of these, WSANOTINITIALISED, indicates that WSAStartup must be called before the gethostname function can be used. We can easily add the WSAStartup and WSACleanup calls to our code. Doing so yields a new version of the source file (unix2win\case2\win4.c):

```
#include <Winsock2.h>

#define BUFFLEN 32

void main(int argc, char **argv)
{
    WORD wVersionRequested;
    WSADATA wsaData;
    char hname[BUFFLEN];

    wVersionRequested = MAKEWORD(2, 2);
    WSAStartup(wVersionRequested, &wsaData);

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);

    WSACleanup();
}
```

Compile this version:

```
C:\tmp\unix2win\case2>cl win4.c /Fewin4 /link /defaultlib:Ws2_32.lib
Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win4.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:win4.exe
/defaultlib:Ws2_32.lib
win4.obj
```

And run it:

```
C:\tmp\unix2win\case2>win4
My name is HN-SDK
```

Okay, this output looks correct. The host name emitted matches the host name of the system on which we ran this example.

We were able to port the gethostname function call with just a few modifications to the source code. Now let's look at porting the MPI calls from Unix to Windows.

Porting to Windows II: MPI

Since gethostname is working, we can turn our attention to moving the MPICH1 calls in our original source file to Windows and Microsoft's version of MPI, MSMPI.

In order to reproduce the examples in this section, you should be running on a Windows Server 2008 system with the HPC Pack installed. These examples will have more meaning if you are using an actual cluster, although that is not strictly required here.

> ***Helpful Hint #4***
>
> To run a program on all the nodes of a cluster, your working directory must be shared among all the nodes.

Let's start our MPI-porting by adding the MPI calls and included MPI header file from our original Unix source file to the source file we developed in our port of gethostname. Here is the updated source file (unix2win\case2\win5.c):

```
#include <Winsock2.h>
#include "mpi.h"

#define BUFFLEN 32

void main(int argc, char **argv)
{
    WORD wVersionRequested;
    WSADATA wsaData;
    char hname[BUFFLEN];

    wVersionRequested = MAKEWORD(2, 2);
    WSAStartup(wVersionRequested, &wsaData);

    MPI_Init(&argc, &argv);

    gethostname(hname, BUFFLEN);
    printf("My name is %s\n", hname);

    MPI_Finalize();

    WSACleanup();
}
```

Add the compilation and link options necessary to bring in the MSMPI header files and link libraries:

```
C:\tmp\unix2win\case2>cl win5.c /Fewin5 /I"C:\Program Files\Microsoft HPC
Pack 2008 SDK\Include" /link /defaultlib:Ws2_32.lib /libpath:"C:\Program
Files\Microsoft HPC Pack 2008 SDK\Lib\amd64\" /defaultlib:msmpifec
/defaultlib:msmpi

Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win5.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:win5.exe
/defaultlib:Ws2_32.lib
"/libpath:C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\amd64"
/defaultlib:msmpifec
/defaultlib:msmpi
win5.obj
```

To take advantage of the MPI calls, run the executable with mpiexec:

```
C:\tmp\unix2win\case2>mpiexec -n 4 win5.exe
My name is HN-SDK
My name is HN-SDK
My name is HN-SDK
My name is HN-SDK
```

From the output of the run, we can tell that only the head node of the Windows cluster was used to run the four process job.

Recall that when we ran this program on a Unix system, we passed a host.list file to mpirun. The host.list file contained the names of the cluster's nodes on which we wanted to run. Could there be a similar option to mpiexec?

> ***Helpful Hint #5***
>
> A number of Windows tools support the '/?' option. This option invokes usage and help information.

Let's see if mpiexec has help information:

```
C:\tmp\unix2win\case2>mpiexec /?
Usage:
mpiexec [options] executable [args] [ : [options] exe [args] : ... ]
mpiexec -configfile <file name>

common options:
-n <num processes>
-env <env var name> <env var value>
-wdir <working directory>
-path <search path ';' separated>
-hosts n host1 [m1] host2 [m2] ... hostn [mn]
-exitcodes - print processes exit codes
-l - prefix output with process rank
-t [filter] - trace the mpi processes
-d [0-3] - print debug output

examples:
mpiexec -n 4 appl
mpiexec -hosts 1 foo master : -n 8 worker

for a complete list of options, run 'mpiexec -help2'
for environment variables list, run 'mpiexec -help3'
```

Although none of those options appears to be what we are looking for, there is an option to get additional help information. Let's try it:

```
C:\tmp\unix2win\case2>mpiexec -help2
mpiexec options:

-configfile filename
 read mpiexec command line from filename.
 The lines of filename are command line sections of the form,
     [options] executable [args]
 which may span across multiple lines by terminating a line with '\'.
 lines beginning with '#' are comments. empty lines are ignored.

-n m, -np m
 launch m processes

-n *, -np *
 launch m processes, where m is the number of available cores.
 the absence of the -n * option has the same meaning.

-machinefile filename
 use a file to list the names of hosts to launch on.
 one host per line optionally followed by number of cores.
 empty lines are ignored. comments to end-of-line are marked with '#'.
 (-n * option uses the sum of cores in the file)

< . . . >
```

The –machinefile option seems to be what we want. To use it, though, we will need a file with the names of the nodes in it.

```
c:\tmp\unix2win\case2>type whost.list
HN-SDK
SDK101
SDK102
SDK103
```

Let's use it:

```
C:\tmp\unix2win\case2>mpiexec -n 4 -machinefile whost.list win5.exe

Aborting: Access denied by node 'HN-SDK'.
This node is a resource managed by the Microsoft HPC Scheduler and
mpiexec was attempting to use it without a scheduled job.
```

Oops! It looks like the system on which we tried to run this executable is not set up to allow access to the other nodes by using mpiexec directly. Fortunately, there is a way to use mpiexec using the Microsoft HPC Job Manager, and we can do this from the command line.

```
C:\tmp\unix2win\case2>job submit /numprocessors:4  /workdir:\\hn-
sdk\tmp\unix2win\case2\ /stdout:win5.out mpiexec win5.exe
Job had been submitted. ID: 165.
```

There were no errors in submission this time. Let's check the output file:

```
C:\tmp\unix2win\case2>type win5.out
My name is SDK101
My name is SDK101
My name is SDK101
My name is SDK101
```

It might be confusing at first that the jobs just ran on one node. It turns out that each node of the cluster on which we are running has four cores (two sockets per node, two cores per socket). So our 4P job ran on a single node.

> ***Helpful Hint #6***
> An incorrect path to the working directory is a common reason a submitted job will fail.

Let's try running on more nodes by requesting more processors:

```
C:\tmp\unix2win\case2>job submit /numprocessors:16
/workdir:\\hn-sdk\tmp\unix2win\case2\ /stdout:win5.out mpiexec win5.exe
Job had been submitted. ID: 166.
```

Before we review the output file, let's check the status of the job using the HPC Job Manager, part of the Microsoft HPC Pack. Look for the job with the ID 166.

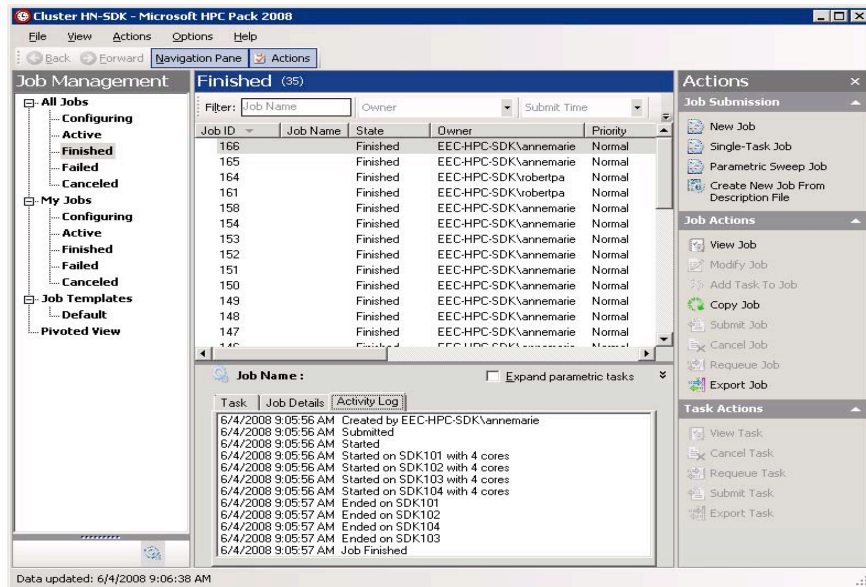


Figure 13: Windows HPC Server 2008 Job Manager

It looks like the job finished. Now we can check the output:

```
C:\tmp\unix2win\case2>type win5.out
My name is SDK101
My name is SDK101
My name is SDK103
My name is SDK101
My name is SDK102
My name is SDK104
My name is SDK103
My name is SDK103
My name is SDK104
My name is SDK101
My name is SDK103
My name is SDK102
My name is SDK104
My name is SDK102
My name is SDK102
My name is SDK104
```

Voila! We have run the job on 16 cores spread across four nodes.

You can also submit a job using the HPC Job Manager interface; refer to the HPC Job Manager Help for details.

MSMPI Debugging

Porting a complex MPI application from Unix to Windows will be more involved than the straightforward port we just walked through. In your porting process, you may encounter a situation where you will need to use an MSMPI debugging tool. One of these tools, the PGI debugger PGDBG, has the ability to debug an MSMPI program running on a Windows cluster.

> ***Helpful Hint #7***
>
> To use PGDBG, you need your environment set up. A quick way to do this is to double-click on the PGI Workstation icon on your desktop.

Let's start by adding the /Zi option to the compilation of the source file we used in the last section. This option causes *cl* to generate debug information:

```
C:\tmp\unix2win\case2>cl /Zi win5.c /Fewin5 /I"C:\Program Files\Microsoft HPC
Pack 2008 SDK\Include" /link  /defaultlib:Ws2_32.lib /libpath:"C:\Program
Files\Microsoft HPC Pack 2008 SDK\Lib\amd64\"  /defaultlib:msmpifec
/defaultlib:msmpi

Microsoft (R) C/C++ Optimizing Compiler Version 15.00.21022.08 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

win5.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/debug
/out:win5.exe
/defaultlib:Ws2_32.lib
"/libpath:C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\amd64"
/defaultlib:msmpifec /defaultlib:msmpi
win5.obj
```

Single-node MPI Debugging

To invoke the PGDBG debugger to debug an MSMPI application, use the -mpi option.

-mpi[:<path_to_mpiexec>]

> The -mpi option can take an optional pathname argument specifying the location of mpiexec, or another similar launcher, when debugging serially. If there are spaces in the path to the launcher, the path must be quoted. You don't need to provide the location of mpiexec if it is already a part of your PATH environment variable.

We'll start by debugging using just the head node (recall that the HPC Job Manager often requires parallel jobs to be submitted through it instead of from the command line; we will tackle that next):

```
C:\tmp\unix2win\case2>pgdbg -text -mpi -n 4 win5.exe
PGDBG 7.2-4 x86-64 (Cluster, 16 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc. All Rights Reserved.
pgdbg: listening for pgserv connections on port 60521

pgdbg [all] 3>
```

Set a breakpoint on the line just after the call to gethostname:

```
pgdbg [all] 3> break "win5.c"@18
[all] (1)breakpoint set at: main line: "win5.c"@18 address: 0x14000107A
1
pgdbg [all] 3>
```

Proceed to the breakpoint:

```
pgdbg [all] 3> cont
([1] New Process)
([2] New Process)
([3] New Process)
Loaded: C:/Windows/system32/ntdll.dll
Loaded: C:/Windows/system32/kernel32.dll
Loaded: C:/Windows/system32/WS2_32.dll
Loaded: C:/Windows/system32/msvcrt.dll
Loaded: C:/Windows/system32/ADVAPI32.dll
Loaded: C:/Windows/system32/RPCRT4.dll
Loaded: C:/Windows/system32/NSI.dll
Loaded: C:/Windows/system32/msmpi.dll
Loaded: C:/Windows/WinSxS/amd64_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.14
34_none_88de292b2fb06019/MSVCR80.dll
Loaded: C:/Windows/system32/MSWSOCK.dll
Loaded: C:/Windows/system32/Secur32.dll
([1.1] New Thread)
([2.1] New Thread)
([3.1] New Thread)
([1.2] New Thread)
([2.2] New Thread)
Loaded: C:/Windows/system32/DNSAPI.dll
Loaded: C:/Windows/System32/winrnr.dll
Loaded: C:/Windows/system32/WLDAP32.dll
Loaded: C:/Windows/system32/PSAPI.DLL
Loaded: C:/Windows/system32/NLAapi.dll
Loaded: C:/Windows/system32/USER32.dll
Loaded: C:/Windows/system32/GDI32.dll
Loaded: C:/Windows/system32/IPHLPAPI.DLL
Loaded: C:/Windows/system32/dhcpcsvc.DLL
Loaded: C:/Windows/system32/WINNSI.DLL
Loaded: C:/Windows/system32/dhcpcsvc6.DLL
Loaded: C:/Windows/system32/IMM32.DLL
Loaded: C:/Windows/system32/MSCTF.dll
Loaded: C:/Windows/system32/LPK.DLL
Loaded: C:/Windows/system32/USP10.dll
Loaded: C:/Windows/system32/napinsp.dll
Loaded: C:/Windows/system32/wshtcpip.dll
Loaded: C:/Windows/system32/credssp.dll
Loaded: C:/Windows/system32/CRYPT32.dll
Loaded: C:/Windows/system32/MSASN1.dll
Loaded: C:/Windows/system32/USERENV.dll
Loaded: C:/Windows/system32/schannel.dll
Loaded: C:/Windows/system32/NETAPI32.dll
Loaded: C:/Windows/system32/pwdssp.dll
Loaded: C:/Windows/system32/msv1_0.dll
Loaded: C:/Windows/system32/cryptdll.dll
([3.2] New Thread)
([0.1] New Thread)
([0.2] New Thread)
Loaded: C:/Windows/system32/ibndprov.dll
Loaded: C:/Windows/system32/complib.dll
Loaded: C:/Windows/system32/mthcau.dll
Loaded: C:/Windows/system32/ibal.dll
Loaded: C:/Windows/system32/rasadhlp.dll
[3.0] Breakpoint at 0x14000107A, function main, file win5.c, line 18
#18:         printf("My name is %s\n", hname);

pgdbg [all] 3.0>
```

Print the value of hname:

```
pgdbg [all] 3.0> p hname
(char *) 0x12FEE0 "HN-SDK"
```

To print the value of hname for all processes, use a process/thread set which has the form “process.thread”. Here, we want thread 0 of all processes so use ‘[*.0]’.

```
pgdbg [all] 3.0> [*.0] p hname
```

The output confirms that we are running all processes on the head node:

```
[0.0] print  hname:
(char *) 0x12FEE0 "HN-SDK"
[1.0] print  hname:
(char *) 0x12FEE0 "HN-SDK"
[2.0] print  hname:
(char *) 0x12FEE0 "HN-SDK"
[3.0] print  hname:
(char *) 0x12FEE0 "HN-SDK"
```

Distributed MPI Debugging—Command Line

The PGI debugger can also debug MPI programs on multiple nodes. Distributed debugging requires the use of two options:

-mpi:<job_submit_command>

The -mpi option provides the job submit command from the command line for distributed debugging. If any path in <job_submit_command> includes spaces, you will need to use quotes. Quote usage varies by shell. Using a bash shell, provide quotes around just the job submit command, -mpi:"…". Using a command prompt, quote the entire option, "-mpi:…".

-pgserv[:<path_to_pgserv.exe>]

The -pgserv option to PGDBG is required to enable distributed debugging.

PGDBG debugs on remote nodes using a remote debug agent called pgserv. The debugger must have access to pgserv on all nodes of the cluster used for the debug session. PGDBG copies the required pgserv executable into the current working directory when needed.

The argument to -pgserv is used to provide the path to pgserv.exe. This argument is not required when pgserv.exe is in the current working directory.

Launch the debugger with the -mpi and -pgserv options to take advantage of the HPC Job Manager:

```
C:\tmp\unix2win\case2>pgdbg -text -pgserv -mpi:job.exe submit /numprocessors:16
/workdir:\\hn-sdk\tmp\unix2win\case2 mpiexec win5.exe
copying C:\PROGRA~1\PGI\win64\7.2-4\bin\pgserv.exe to current directory
        1 file(s) copied.
PGDBG 7.2-4 x86-64 (Cluster, 16 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc. All Rights Reserved.
pgdbg: listening for pgserv connections on port 52438
Job had been submitted. ID: 167.
```

Set a breakpoint as we did before, just after the call to gethostname:

```
pgdbg [all] 15> break "win5.c"@18
[all] (1)breakpoint set at: main line: "win5.c"@18 address: 0x14000107A
1
```

And proceed:

```
pgdbg [all] 15> cont
([1] New Process)
([2] New Process)
([3] New Process)
([4] New Process)
([5] New Process)
([6] New Process)
([7] New Process)
([8] New Process)
([9] New Process)
([10] New Process)
([11] New Process)
([12] New Process)
([13] New Process)
([14] New Process)
([15] New Process)
Loaded: C:/Windows/system32/ntdll.dll
Loaded: C:/Windows/system32/kernel32.dll
Loaded: C:/Windows/system32/WS2_32.dll
Loaded: C:/Windows/system32/msvcrt.dll
Loaded: C:/Windows/system32/ADVAPI32.dll
Loaded: C:/Windows/system32/RPCRT4.dll
Loaded: C:/Windows/system32/NSI.dll
Loaded: C:/Windows/system32/msmpi.dll
Loaded:
C:/Windows/WinSxS/amd64_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.0.50727.14
34_none_88de292b2fb06019/MSVCR80.dll
Loaded: C:/Windows/system32/MSWSOCK.dll
Loaded: C:/Windows/system32/Secur32.dll
([1.1] New Thread)
([2.1] New Thread)
([3.1] New Thread)
([4.1] New Thread)
([5.1] New Thread)
([6.1] New Thread)
([7.1] New Thread)
([8.1] New Thread)
([9.1] New Thread)
([10.1] New Thread)
([11.1] New Thread)
([12.1] New Thread)
([13.1] New Thread)
([14.1] New Thread)
([15.1] New Thread)
([1.2] New Thread)
([2.2] New Thread)
([3.2] New Thread)
([4.2] New Thread)
([5.2] New Thread)
([6.2] New Thread)
([7.2] New Thread)
([8.2] New Thread)
([9.2] New Thread)
([10.2] New Thread)
([11.2] New Thread)
([12.2] New Thread)
([13.2] New Thread)
([14.2] New Thread)
([15.2] New Thread)
([0.1] New Thread)
Loaded: C:/Windows/system32/DNSAPI.dll
Loaded: C:/Windows/System32/winrnr.dll
Loaded: C:/Windows/system32/WLDAP32.dll
Loaded: C:/Windows/system32/PSAPI.DLL
Loaded: C:/Windows/system32/NLAapi.dll
Loaded: C:/Windows/system32/USER32.dll
Loaded: C:/Windows/system32/GDI32.dll
Loaded: C:/Windows/system32/IPHLPAPI.DLL
Loaded: C:/Windows/system32/dhcpcsvc.DLL
Loaded: C:/Windows/system32/WINNSI.DLL
Loaded: C:/Windows/system32/dhcpcsvc6.DLL
```

```
Loaded: C:/Windows/system32/IMM32.DLL
Loaded: C:/Windows/system32/MSCTF.dll
Loaded: C:/Windows/system32/LPK.DLL
Loaded: C:/Windows/system32/USP10.dll
Loaded: C:/Windows/system32/napinsp.dll
Loaded: C:/Windows/system32/wshtcpip.dll
Loaded: C:/Windows/system32/credssp.dll
Loaded: C:/Windows/system32/CRYPT32.dll
Loaded: C:/Windows/system32/MSASN1.dll
Loaded: C:/Windows/system32/USERENV.dll
Loaded: C:/Windows/system32/schannel.dll
Loaded: C:/Windows/system32/NETAPI32.dll
Loaded: C:/Windows/system32/msv1_0.dll
Loaded: C:/Windows/system32/cryptdll.dll
([0.2] New Thread)
Loaded: C:/Windows/system32/ibndprov.dll
Loaded: C:/Windows/system32/complib.dll
Loaded: C:/Windows/system32/mthcau.dll
Loaded: C:/Windows/system32/ibal.dll
Loaded: C:/Windows/system32/rasadhlp.dll
[15.0] Breakpoint at 0x14000107A, function main, file win5.c, line 18
#18:         printf("My name is %s\n", hname);

pgdbg [all] 15.0>
```

Print hname for the current process:

```
pgdbg [all] 15.0> p hname
(char *) 0x12FEE0 "SDK104"
```

Then all the processes:

```
pgdbg [all] 15.0> [*.0] p hname
[0.0] print  hname:
(char *) 0x12FEE0 "SDK101"
[1.0] print  hname:
(char *) 0x12FEE0 "SDK101"
[2.0] print  hname:
(char *) 0x12FEE0 "SDK101"
[3.0] print  hname:
(char *) 0x12FEE0 "SDK101"
[4.0] print  hname:
(char *) 0x12FEE0 "SDK102"
[5.0] print  hname:
(char *) 0x12FEE0 "SDK102"
[6.0] print  hname:
(char *) 0x12FEE0 "SDK102"
[7.0] print  hname:
(char *) 0x12FEE0 "SDK102"
[8.0] print  hname:
(char *) 0x12FEE0 "SDK103"
[9.0] print  hname:
(char *) 0x12FEE0 "SDK103"
[10.0] print  hname:
(char *) 0x12FEE0 "SDK103"
[11.0] print  hname:
(char *) 0x12FEE0 "SDK103"
[12.0] print  hname:
(char *) 0x12FEE0 "SDK104"
[13.0] print  hname:
(char *) 0x12FEE0 "SDK104"
[14.0] print  hname:
(char *) 0x12FEE0 "SDK104"
[15.0] print  hname:
(char *) 0x12FEE0 "SDK104"
```

Distributed MPI Debugging—GUI

Using the PGDBG GUI for distributed debugging is as simple as removing the -text option from the pgdbg command we have been using:

```
C:\tmp\unix2win\case2>pgdbg -pgserv -mpi:job.exe submit
/numprocessors:16 /workdir:\\hn-sdk\tmp\unix2win\case2 mpiexec win5.exe
```

If the source file does not appear in the source pane on start-up, use the file drop-down box in the middle of the GUI to select "win5.c". Set a breakpoint on line 18 after the call to gethostname and click the “Cont” button.

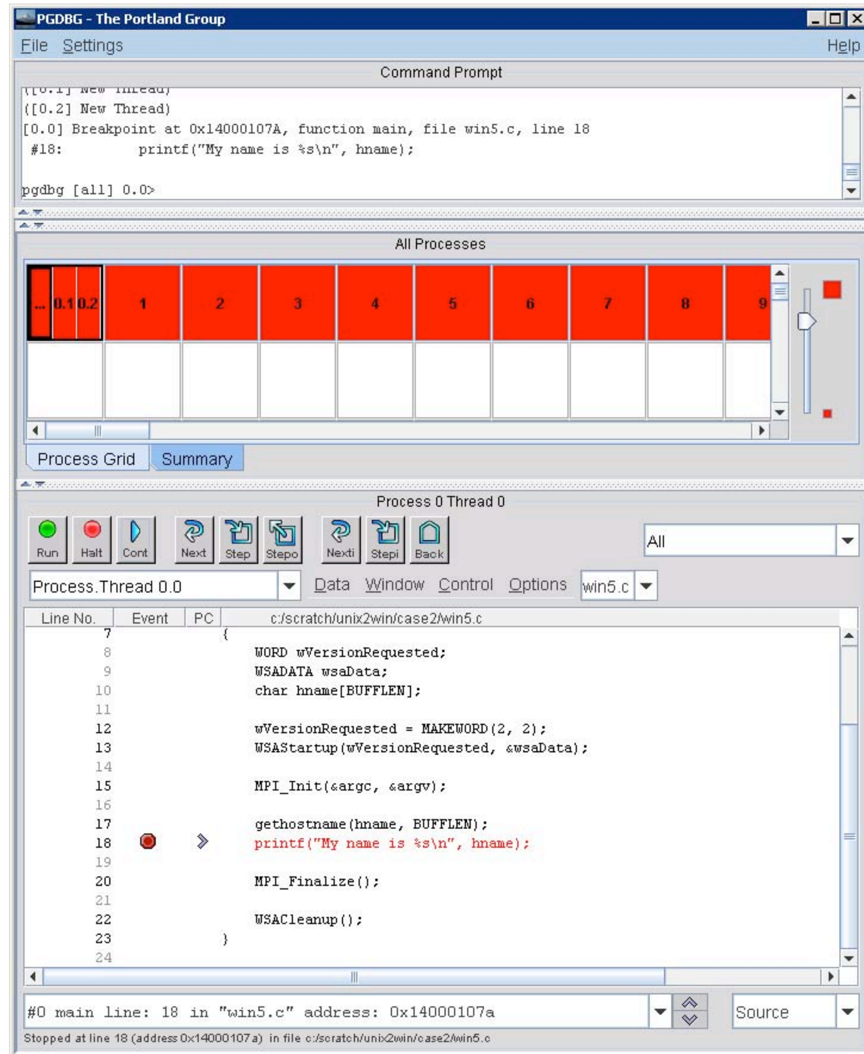


Figure 14 MSMPI debugging with PGDBG

Debug as you normally would.

MPI in SUA

There is no MPI support in the Microsoft Subsystem for UNIX-based Applications.

Case Study 3: Porting Shared Objects

Static libraries on Windows (.lib files) work much the same as static libraries on Unix (.a files). There are important differences, however, between how shared object libraries on Unix (.so files) and their conceptual counterpart on Windows, dynamically-linked libraries or DLLs (.dll files), behave.

This case study will walk through the process of porting a Unix program linked against two shared objects to a Windows program linked against two DLLs. The example has been simplified from a real-world situation in order to highlight the differences in how Unix shared objects and Windows DLLs handle global data.

The case study concludes with a demonstration of how to port the same Unix program to Windows SUA.

Background

Before getting into a comparison of Windows DLLs and Unix shared objects, it might help to review the differences between Windows static and Windows dynamic libraries. These differences are similar to those between Unix static and Unix shared object libraries. Both static and dynamic libraries are used when resolving external references for linking an executable. When linking with a static library, the code needed from the library is incorporated into the executable. When linking with a dynamic library, external references are typically 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. Instead, the DLL is loaded when the executable that needs it is run.

> ***Helpful Hint #8***
>
> When a Windows executable requires a DLL at runtime, the system must be able to find the DLL. The system looks in its default runtime directories first, which is where it typically finds system DLLs. Putting the directory containing the DLL your application needs in your PATH environment variable will ensure that the system will find that DLL.

Starting Point: Unix

In this example, global data in the form of a Fortran common block will be shared between the main program and two shared objects. Here is the code for the Fortran main program (unix2win\case3\unix\prog.f90):

```
program prog
  external init_data, print_data
  common /a_common/ data
  integer data
  call print_data()
  call init_data(11)
  call print_data()
end
```

Here is the code for one of the shared objects (unix2win\case3\unix\a.f90):

```
subroutine init_data(i)
  common /a_common/ data
  integer data
  integer i
  data = i
  print *, " init_data:", data
end
```

The other shared object contains this code (unix2win\case3\unix\b.f90):

```
subroutine print_data()
  common /a_common/ data
  integer data
  print *, "print_data:", data
end
```

On Unix, here's how you would build the two .so files and the executable:

```
% cd unix2win/case3/unix
% pgf95 -fpic -shared a.f90 -o a.so
% pgf95 -fpic -shared b.f90 -o b.so
% pgf95 prog.f90 -o prog a.so b.so
```

If $LD_LIBRARY_PATH does not include the current directory, set it now. This environment variable directs the system to location of the .so files.

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":.
```

When you run the program, you should see output similar to the following:

```
% prog
 print_data:            0
  init_data:           11
 print_data:           11
```

Be sure to take note of the program output. We will expect to see the same print-out on Windows.

Porting Unix Shared Objects to Native Windows

Now we will move this Fortran code to Windows using the PGI Workstation compilers. Let's try building and running the code without any modification. Maybe it will just work.

```
PGI$ cd unix2win/case3/unix
PGI$ make clean
PGI$ pgf95 -Bdynamic -Mmakedll a.f90 -o a.dll
PGI$ pgf95 -Bdynamic -Mmakedll b.f90 -o b.dll
PGI$ pgf95 -Bdynamic prog.f90 -o prog -defaultlib:a.lib -defaultlib:b.lib
LINK : fatal error LNK1104: cannot open file 'a.lib'
```

The link failed. When a program needs something in a DLL, it typically links against the DLL's import library, usually a .lib file. The error we see above indicates that the import library a.lib was not found. Let's check what was built when a.f90 was compiled:

```
PGI$ ls a.*
a.dll  a.dwf  a.f90
```

So the DLL exists but its import library does not. The import library contains all the symbols a DLL exports. The primary reason an import library is not created when a DLL is built is because the DLL did not export any symbols.

One of the ways to export symbols from a DLL is to modify the source code. For Fortran, the DEC$ ATTRIBUTES extension DLLEXPORT is provided to mark a symbol for export. For example:

```
!DEC$ ATTRIBUTES DLLEXPORT :: fortranFunc
```

For C/C++, Microsoft provides the storage class modifier __declspec(dllexport) to accomplish the same thing. For example:

```
int __declspec(dllexport) cFunc()
```

In our example, we will use !DEC$ ATTRIBUTES DLLEXPORT to mark the init_data and print_data functions for export:

```
subroutine init_data(i)
!DEC$ ATTRIBUTES DLLEXPORT :: init_data
  common /a_common/ data
  integer data
  integer i
  data = i
end

subroutine print_data()
!DEC$ ATTRIBUTES DLLEXPORT :: print_data
  common /a_common/ data
  integer data
  print *, "data:", data
end
```

Build again:

```
PGI$ pgf95 -Bdynamic -Mmakedll a.f90 -o a.dll
   Creating library a.lib and object a.exp
PGI$ pgf95 -Bdynamic -Mmakedll b.f90 -o b.dll
   Creating library b.lib and object b.exp
PGI$ pgf95 -Bdynamic prog.f90 -o prog -defaultlib:a.lib -defaultlib:b.lib
```

There were no build errors, so we are ready to run:

```
PGI$ prog
 print_data:            0
  init_data:           11
 print_data:            0
```

The program runs without errors but the output does not match the Unix output. You can see that the init_data call was made, and that the variable data was set to 11, but for some reason the value of that variable is 0 when the next call to print_data is made.

This situation illustrates a fundamental difference between dynamically linked libraries on Windows and Unix static, Unix shared object, and Windows static libraries. DLLs treat global data in a different way than the rest of these library types.

Global data in static libraries and Unix shared objects is automatically accessible to other objects linked into an executable. This model works because global data is shared; variables with the same name in multiple images resolve to the same physical data. Here is a graphical depiction of how this model is implemented with .so files on Unix:

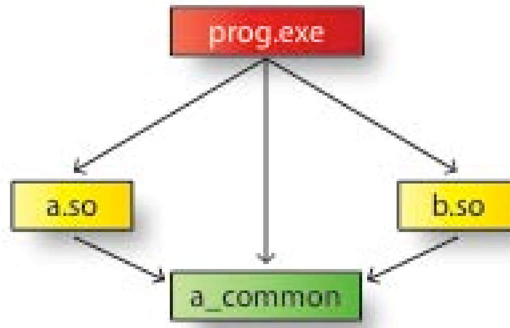


Figure 15: Unix Shared Object Model

In contrast, when a DLL is loaded on Windows, space is set aside for its global data. This space is not shared with the space set aside for data by other image files. Each image gets its own separate copy of the data; this situation is modeled in Figure 16.

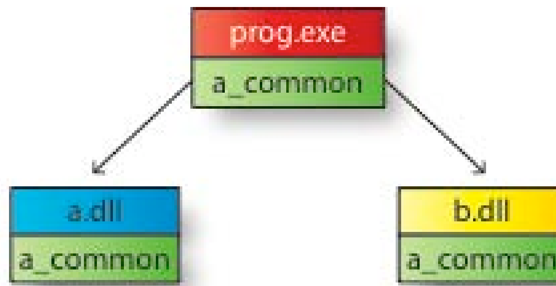


Figure 16: Windows DLL Model without Import/Export

Global data in a DLL can be accessed from outside the DLL only if the DLL exports the data and the image that uses the data imports it. The resulting model looks like this:

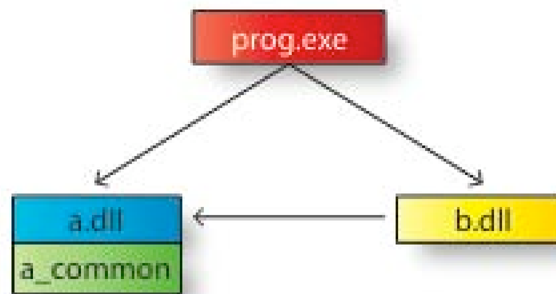


Figure 17: Windows DLL Model with Import/Export

The same methods used to export functions can be used to export data.

In Fortran:

```
!DEC$ ATTRIBUTES DLLEXPORT :: /fData/
```

In C:

```
float __declspec(dllexport) cData;
```

Both data and functions are imported using a similar method.

In Fortran:

```
!DEC$ ATTRIBUTES DLLIMPORT :: fortranFunc
!DEC$ ATTRIBUTES DLLIMPORT :: /fData/
```

In C:

```
extern int __declspec(dllimport) cFunc();
extern float __declspec(dllimport) cData;
```

Using DLLEXPORT and DLLIMPORT for the global data, our code now looks like:

```
program prog
  common /a_common/ data
  integer data
!DEC$ ATTRIBUTES DLLIMPORT :: /a_common/, init_data, print_data

  call print_data()
  call init_data(11)
  call print_data()

end

subroutine init_data(i)
!DEC$ ATTRIBUTES DLLEXPORT :: init_data
  common /a_common/ data
  integer data
  integer i
!DEC$ ATTRIBUTES DLLEXPORT :: /a_common/
  data = i
  print *, " init_data:", data
end

subroutine print_data()
!DEC$ ATTRIBUTES DLLEXPORT :: print_data
  common /a_common/ data
  integer data
!DEC$ ATTRIBUTES DLLIMPORT :: /a_common/
  print *, "print_data:", data
end
```

We have to modify the commands we use to build b.dll because it now must link against a.lib to satisfy its imports.

```
PGI$ pgf95 -Bdynamic -Mmakedll a.f90 -o a.dll
   Creating library a.lib and object a.exp
PGI$ pgf95 -Bdynamic -Mmakedll b.f90 -o b.dll -defaultlib:a.lib
   Creating library b.lib and object b.exp
PGI$ pgf95 -Bdynamic prog.f90 -o prog -defaultlib:a.lib -defaultlib:b.lib
```

Run the program.

```
PGI$ prog
 print_data:           0
  init_data:          11
 print_data:          11
```

Success! This output matches that produced by the Unix program. This shared object port to native Windows is complete.

Porting Unix Shared Objects to Windows SUA

SUA supports the traditional Unix shared objects model. The 32-bit SUA linker can create .so libraries, and the port from Unix is straightforward. The 64-bit SUA linker has limitations that prevent it from producing .so libraries, so shared objects in SUA can only be 32-bit.

We can take the source for the a.f90, b.f90 and prog.f90 files that compile and run on Unix and compile and run them on SUA without any modification. Some of the compilation options differ when using the PGI compilers on SUA vs. native Windows; we will walk through these steps.

Begin by setting the SUA environment (C shell) to use the 32-bit PGI compilers:

```
% setenv PGI /opt/pgi
% setenv PATH ".:$PGI/sua32/7.2-4/bin:$PATH"
% setenv PGIBIN "$PGI/sua32/7.2-4/bin"
```

Compile the .so files and link the main program against them:

```
% cd unix2win/case3/sua
% pgf95 -shared -Mnostartup -Mnostdlib a.f90 -o a.so
% pgf95 -shared -Mnostartup -Mnostdlib b.f90 -o b.so
% pgf95 prog.f90 -o prog a.so b.so
prog.f90:
```

Set LD_LIBRARY_PATH so it includes the current directory. This environment variable directs the system to location of the .so files.

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":.
```

Run:

```
% prog
 print_data:           0
  init_data:          11
 print_data:          11
```

The output is as expected.

Conclusion

It is not unusual to be daunted at the beginning of a porting project. Porting an application from a Unix-like environment to the Windows platform is going to take effort just like any other port. Planning the port at the outset will make the process proceed more smoothly. The build environment options discussed—native Windows vs. SUA, command-line vs. IDE—give you choices so you can develop a migration path that fits your needs and moves at your pace. Many of the significant issues you may encounter moving to Windows were addressed, and tips were given to help with some of the smaller, yet no less potentially frustrating, workaday tasks. The case studies' step-by-step examples illustrated common stumbling blocks and procedures for how to solve these and other Windows porting issues. Taken together, this information should help ease your transition to Windows.

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of The Portland Group Incorporated.

© 2008 The Portland Group Incorporated. All rights reserved.

Microsoft, Visual Studio, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

PGF95, PGF90, and PGI Unified Binary are trademarks; and PGI, PGI CDK, PGHPF, PGF77, PGCC, PGC++, PGI Visual Fortran, PVF, Cluster Development Kit, PGPROF, PGDBG, and The Portland Group are registered trademarks of The Portland Group Incorporated, and STMicroselectronics company.

All other marks are the property of their respective owners.

0908

References

This paper was developed using information from several sources.

Details on the use of gethostname, WSACleanup and WSAStartup were obtained from the Microsoft Visual Studio 2008 documentation available with installations of VS 2008. Similar documentation is available from the Microsoft Developer Network (MSDN).

The Microsoft HPC Pack for Windows Server 2008 contains its own Help documentation which proved useful in learning to use the HPC Job Manager.

Two documents from The Portland Group were relied upon throughout the development of the case studies:

- The PGI User's Guide (http://www.pgroup.com/doc/pgiug.pdf) describes the general features and usage guidelines for all PGI compilers, and lists in detail various available compiler options in a user's guide format.
- The PGI Visual Fortran User's Guide (http://www.pgroup.com/doc/pvfug.pdf) contains much of the same information as the PGI User's Guide but is targeted to users of Visual Studio.