

Tools for the Classic HPC Developer

The Portland Group

Published: v2.0 September 2008

Abstract

Fortran was one of the first high-level computer programming languages, and has been in use for over 50 years. During that time, a huge reservoir of HPC applications has been developed in Fortran. The Fortran language has evolved to support both performance-oriented programming and modern best practices features in programming languages and parallel computing, and many HPC applications have incorporated components in C and C++ as well. As a result, there is an incredible range and diversity of HPC applications in use today, most of which were developed on UNIX-heritage HPC and cluster platforms. This paper presents the compiler and tools resources available and shows practical examples of porting such codes for use on Windows HPC Server 2008 clusters.

Contents

Introduction	1
Fortran and Windows HPC Server 2008	1
SIMD Vectorization for x64 Processors	1
Optimizing Fortran/C/C++ Compilers for Windows	3
Timer Function Calls in HPC Applications	4
Installing PGI Workstation Compilers and Tools	6
Porting the OpenMP NAS Parallel Benchmarks to Windows	10
Getting Started.....	10
Using Make to Build the FT Benchmark.....	11
Tweaking the Timer Function	14
Object File Filename Extensions	16
A Successful Build of NAS FT	17
Parallel Execution.....	19
Porting the MPI NAS Parallel Benchmarks to Windows.....	22
Debugging MPI and OpenMP Programs	29
Building the Linpack HPC Benchmark	42
Step 1: Download the HPL benchmark	42
Step 2: Create the Makefile	42
Step 3: Configure the Makefiles	42
Step 4: Build the Benchmark	46
Step 5: Run the Benchmark	46
Step 6: Review the Results.....	48
Step 7: Tuning	50
Tuning the Input Data	50
Enable Compiler Optimizations	52
An Improved BLAS.....	53
Increase Number of Processors.....	54
Conclusion	56
References.....	57

Introduction

Fortran and Windows HPC Server 2008

Many HPC applications are written in early FORTRAN dialects and continue to serve their function without need for significant modification; they deliver good performance on a continuing basis as the state-of-the-art in CPU and compiler technology marches forward. Other HPC applications were developed using FORTRAN IV or FORTRAN 77, and have continued to actively evolve for years and sometimes decades.

These applications often have some components that rely on legacy language features, and other components that are thoroughly modern in their use of Fortran 90 pointers, derived types, array syntax and modular programming. Still other modern HPC Fortran applications have been developed only recently, and use Fortran features and programming styles that would be all but unrecognizable to a classic Fortran programmer.

Many of the performance-oriented compilers, tools and libraries you are accustomed to using on traditional HPC platforms are now available on Windows HPC Server 2008 from Microsoft and its partners. Whether you need to port a simple legacy Fortran application for use on Windows, or you are undertaking a major parallel application port from a traditional HPC platform to Windows HPC Server 2008, this white paper will walk you through the steps and resources available to get your applications up and running quickly.

Included are overviews of compiler technologies required to maximize performance of compute-intensive applications on Windows, available resources for parallel programming using OpenMP and MPI, tools available for debugging and profiling Windows cluster applications, and an introduction to Fortran programming within Microsoft Visual Studio 2008. Most of these concepts are presented through simple step-by-step tutorials you can easily reproduce using the companion example codes included with this white paper.

SIMD Vectorization for x64 Processors

Windows HPC Server 2008 is supported on systems based on 64-bit x86 (x64) processors including the AMD Opteron and Intel Xeon EM64T families of processors. Both families of processors now support Streaming SIMD Extensions (SSE) instructions for floating-point and integer vector computations, incorporate multiple CPU cores, and have complex hierarchical cache-based memory systems.

Vectorization is an extremely important compiler technology for modern Intel and AMD x64 processors. Vectorization by a programmer is the process of writing or modifying algorithms and loops to enable or maximize generation of x64 packed SSE instructions by a compiler. Vectorization by a compiler is identifying and transforming loops to use packed SSE arithmetic instructions which operate on more than one data element per instruction.

SSE instructions process up to 128 bits of data per instruction—operating on either two 64-bit operands or four 32-bit operands simultaneously. Each operand contains two 64-bit double-precision values or four 32-bit single-precision values. The result of the instruction is also a 128-bit packed result, of the same data type as the inputs. Figure 1 below gives a logical representation of how packed double-precision operands are stored and computed in SSE registers.

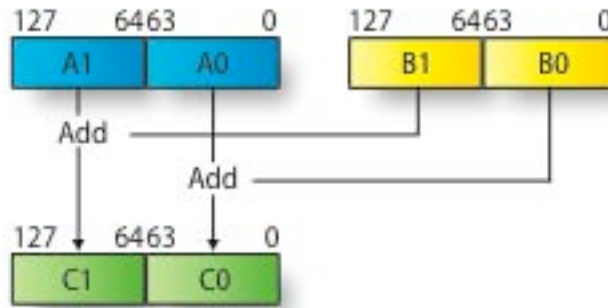


Figure 1: Double-precision packed SSE instructions on x64 processors

Vectorization on x64 processors can result in speed-ups of a factor of two or more on loops that operate on single-precision data, and large percentage speed-ups on loops that operate on double-precision data. For floating-point compute-intensive loop-based applications, it is not uncommon to see 20%–50% speed-ups of the entire application on an x64 processor if most or all of the time-consuming loops can be vectorized.

Consider an abbreviated version of the standard Basic Linear Algebra Subroutines (BLAS) library routine to multiply the elements of one vector by a scalar, and add the results to a second vector:

```

subroutine saxpy(n,da,dx,incx,dy,incy)
doubleprecision dx(1),dy(1),da
integer i,incx,incy,ix,iy,m,mp1,n
. . .
do 30 i = 1,n
    dy(i) = dy(i) + da*dx(i)
30 continue
return
end

```

When compiled using an x64 compiler without vectorization enabled, a simple scalar loop is generated to perform one floating point add and one floating-point multiply per iteration using 7 assembly instructions:

```

.LB1_319:
    movlpd    -8(%edx,%eax,8), %xmm1
    mulsd    %xmm0, %xmm1          # One scalar DP multiply
    addsd    -8(%esi,%eax,8), %xmm1 # One scalar DP add
    movlpd    %xmm1, -8(%esi,%eax,8)
    incl     %eax
    cmpl     %ecx, %eax
    jl      .LB1_319

```

When compiled with vectorization enabled, an unrolled vector SIMD loop is generated that performs two floating point operations per packed SSE instruction, and 16 total double-precision floating-point operations per loop iteration, in just 21 assembly instructions:

```

.LB1_518:
    movapd   %xmm0, %xmm1
    movapd   (%edi,%edx), %xmm2
    movapd   16(%edx,%edi), %xmm3
    subl     $8, %eax
    mulpd    %xmm1, %xmm2          # Two scalar DP multiplies
    mulpd    %xmm1, %xmm3          # Two scalar DP multiplies
    addpd    (%esi,%edx), %xmm2    # Two scalar DP adds
    movapd   %xmm2, (%esi,%edx)
    movapd   32(%edx,%edi), %xmm2
    addpd    16(%edx,%esi), %xmm3   # Two scalar DP adds
    mulpd    %xmm1, %xmm2          # Two scalar DP multiplies
    movapd   %xmm3, 16(%edx,%esi)
    mulpd    48(%edx,%edi), %xmm1   # Two scalar DP multiplies
    addpd    32(%edx,%esi), %xmm2   # Two scalar DP adds
    movapd   %xmm2, 32(%edx,%esi)
    addpd    48(%edx,%esi), %xmm1   # Two scalar DP adds
    movapd   %xmm1, 48(%edx,%esi)
    addl     $8, %ecx
    addl     $64, %edx
    testl    %eax, %eax
    jg      .LB1_518

```

The vectorized loop certainly looks much more efficient, when viewed strictly according to the ratio of floating-point operations to total assembly instructions. In fact, when the LINPACK 100 benchmark from which this kernel is extracted is compiled with vectorization enabled, performance improves by over 65% on an AMD Opteron processor.

Optimizing Fortran/C/C++ Compilers for Windows

Vectorization is only one of a number of advanced compiler optimizations that can improve the performance of compute-intensive applications. Extracting maximum performance from floating-point and array-based Fortran, C and C++ applications on x64 processors requires performance-oriented compilers enabled with the following technologies:

- **Loop vectorization** for automatic generation of packed SSE (SIMD) instructions
- **Alternate code generation** to maximize cache-aligned accesses and enable use of non-temporal load and store instructions

- **Software prefetching** to minimize the effects of main memory latency
- **Function inlining** to minimize overhead and expose optimization opportunities
- **Interprocedural analysis** to enable optimization across function boundaries
- **Profile-feedback optimization** to minimize branching penalties, optimize code layout, and optimize operations involving input constants
- **Autoparallelization and OpenMP** support for multi-core optimization
- **Compiler directives** to enable potentially unsafe optimizations under programmer control

The simple SIMD vectorization example above shows the significant performance improvements you can obtain using an optimizing compiler that takes advantage of the latest features of modern x64 processors. On Windows HPC Server 2008, there are two commonly-used 64-bit optimizing compiler packages that support SSE vectorization and most of the other features listed above:

- **PGI® Workstation optimizing/parallel Fortran/C/C++** compilers for Intel and AMD x64 processors can be used from within a provided UNIX-like shell environment, and are also available fully integrated with Microsoft Visual Studio 2008 (*PGI Visual Fortran®*)—see <http://www.pgroup.com> to download a free trial version.
- **Intel optimizing/parallel Fortran/C/C++** compilers for Intel x64 processors can be used from a command interface using options compatible with the traditional Windows DOS-style shell command environment, and are also available fully integrated with Microsoft Visual Studio 2008 (*Intel Visual Fortran*)—see <http://developer.intel.com/> to download a free trial version.

In addition to being usable from Microsoft Visual C++ programs, the MSMPI libraries included with Microsoft Windows HPC Server 2008 are pre-configured for use with both the PGI and Intel optimizing Fortran compilers. All three compiler packages (MS, PGI, Intel) support full native OpenMP 2.5 shared memory parallel programming (<http://www.openmp.org/>), and have varying degrees of support for debugging MPI, OpenMP and hybrid MPI+OpenMP parallel programs. The PGI and Intel Fortran compilers include extensive support for legacy Fortran programming features (MIL-STD extensions, DEC structures and unions, LIB3F functions, Cray POINTERS, etc) in addition to support for full Fortran 95 and some elements of Fortran 2003.

In the tutorials that follow, we will leverage the UNIX-compatible development environment that comes bundled with the PGI Workstation package to show how simple it can be to migrate traditional HPC OpenMP and MPI programs to Windows HPC Server 2008.

Timer Function Calls in HPC Applications

Many HPC applications include some facility for measuring runtime performance or self-profiling, usually in the form of calls to timer functions. In general, timing routines can be a stumbling block to porting HPC applications from platform-to-platform, and Windows HPC Server 2008 is no exception. Table 1 lists several common timer functions you may encounter and suggests workarounds for addressing calls to these functions when porting to Windows:

<i>Timer Function</i>	<i>Source</i>	<i>Moving to Windows</i>
ctime	LIB3F	Poor accuracy; replace using system_clock
etime	LIB3F	Poor accuracy; replace using system_clock
secnds	LIB3F	Poor accuracy; replace using system_clock
gettimeofday	UNIX C Library	Not available in Windows C Library; replace using system_clock
cpu_clock	F95 Intrinsic	Available by default in pgf95 and ifort
system_clock	F90 Intrinsic	Available by default in pgf95 and ifort
omp_get_wtime	OpenMP intrinsic	Available by default in pgf95 and ifort
MPI_Wtime	MPI intrinsic	Supported as part of MSMPI

Table 1" Timer functions

The functions ctime, etime, secnds and several similar legacy timing functions are in fact supported by the PGI and Intel Fortran compilers. However, in some cases the timer resolution is poor (as coarse as 1 second). In general, it is advisable to replace calls to these functions with calls to standard Fortran 90/95 intrinsics. The system_clock function introduced in Fortran 90 is used to measure wall clock time, and the cpu_clock function introduced in Fortran 95 to measure CPU time.

The fragment below shows how to use system_clock within an F90/F95 program unit.

```

. . .
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)
time = (clock1 - clock0) / real(hz)
. . .

```

If you have an application that links in a wall clock timer function which is written in C and calls gettimeofday, there is no equivalent function in the Windows C Library. The best solution is to replace calls to the timer using one of the Fortran intrinsics, but it's also straightforward to find implementations of gettimeofday for Windows using a web search.

Installing PGI Workstation Compilers and Tools

The PGI Workstation command-level Fortran, C and C++ compilers, debugger and profiler look and feel exactly like their counterparts you may have used on Linux clusters. However, they are built on top of and leverage the Microsoft Open Tools linker, libraries and header files to enable you to create native Windows applications that can be deployed on any Windows system. Equally important for our purposes here, the PGI distribution for Windows includes a BASH shell and utilities that allow you to leverage the command-line expertise and utilities you are accustomed to using on UNIX or Linux HPC systems—make, vi, sed, grep, awk, shell scripts, etc—right from your Windows desktop.

You can obtain a free trial of the PGI compilers by visiting the PGI website at <http://www.pgroup.com/>. You will be asked to register prior to downloading the software. Once you have registered, log in to the PGI website. You will be presented with a menu of options, the first of which is a link to the PGI downloads page. Click on that link, and you should see the PGI downloads page which looks as follows:

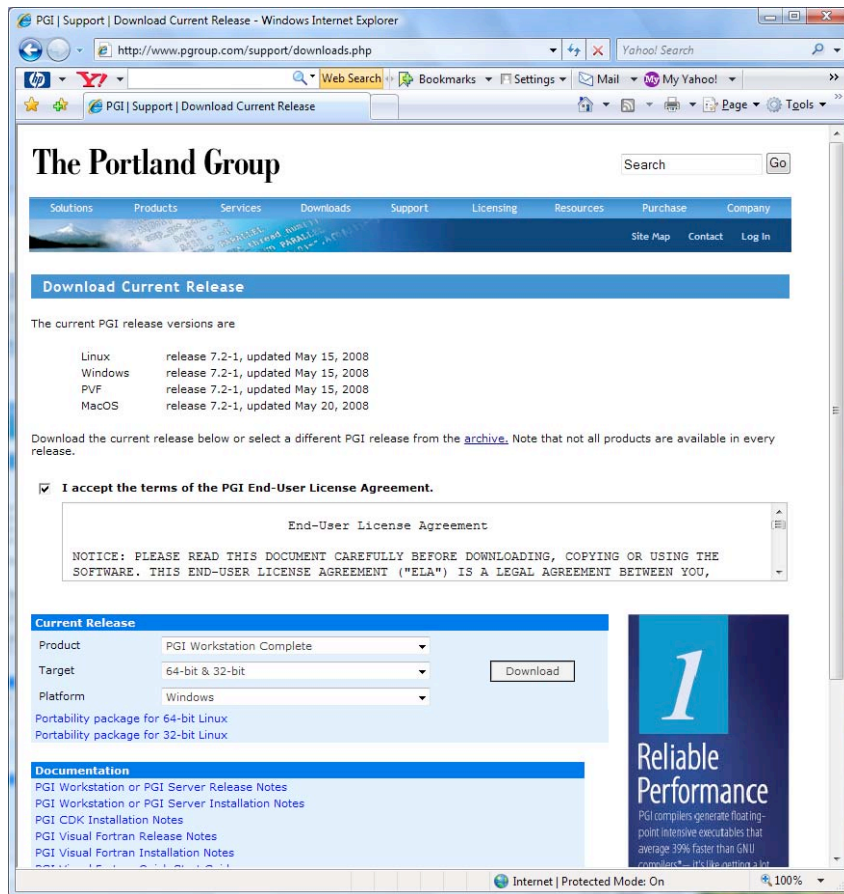


Figure 2: PGI software download website

Simply assent to the license agreement, set **Target** to “64-bit & 32-bit” for 64-bit systems or “32-bit only” if you intend to install on a 32-bit system. Set **Platform** to “Windows”, and click the **Download** button.

Once the download completes, simply double-left-click on the downloaded .exe file and installation will begin. When you are asked if the installer should create a PGI Workstation desktop icon shortcut, and whether to install the optional ACML math libraries, choose yes. We will use these in the tutorial sections below. If you get a pop-up error box with the message “This product cannot be installed on this OS”, you are probably trying to install the 64-bit version of PGI Workstation on a 32-bit operating system. Re-download the 32-bit-only version and try again.

Once you’ve installed the software, double-left-click on the PGI Workstation desktop icon to bring up a command window and issue the command to retrieve the host ID of your system:

```
PGI$ lmutil lmhostid
lmutil – Copyright (c) 1989–2007 Macrovision Europe Ltd. and/or Macrovision
Corporation. All Rights Reserved.
The FLEXnet host ID of this machine is “444553544200 00904bef0a2b 000fb07754d4”

Only use ONE from the list of hostids
PGI$
```

Back in the browser window that is logged in to the PGI website, click on **My Account** in the upper right corner, and then click on the **Create trial keys** link in the middle of the page. Check the box to assent to the license agreement, and enter your FLEXnet host ID in the appropriate text box:

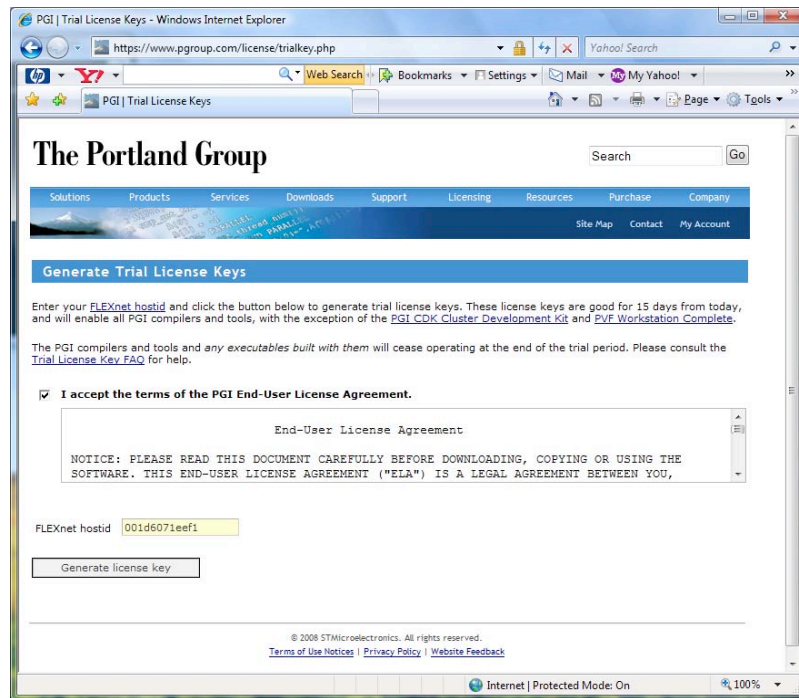


Figure 3: PGI trial license keys website

When the license keys are presented in your browser, click on **Save trial keys as a file on my computer** and save to the file C:\Program Files\PGI\license.dat:

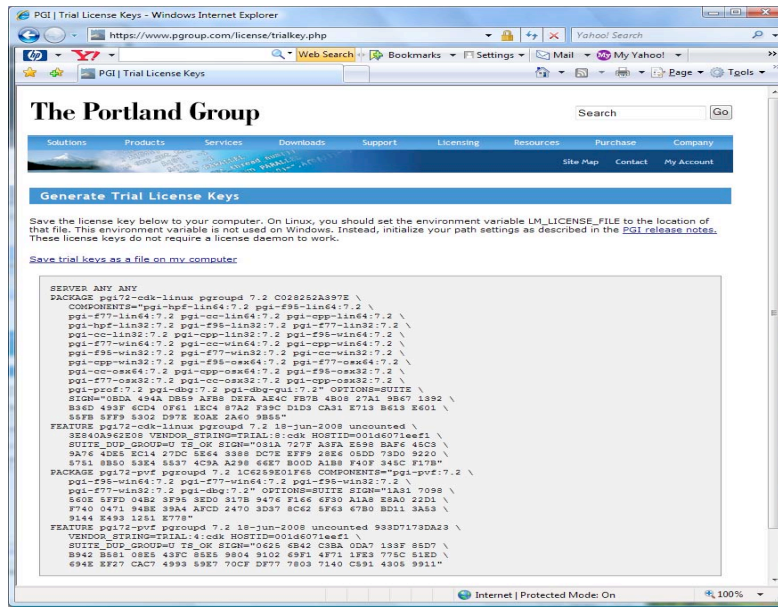


Figure 4: PGI trial license keys website—sample keys

Once you have PGI Workstation installed, you can bring up a BASH shell by double-left-clicking on the PGI Workstation desktop icon. You now have access to over 100 commands and utilities including but not limited to:

awk	date	ftp	make	pgdbg	sed
bc	diff	grep/egrep	more/less	pgf77	tar/untar
cat	du	gzip/gunzip	mv	pgf95	touch
cksum	env	kill	pgcc	pgprof	vi
cp	find	ls	pgcpp	rm/rmdir	wc

Most of the commands as well as the PGI compilers and tools have extensive documentation in the form of UNIX man pages:

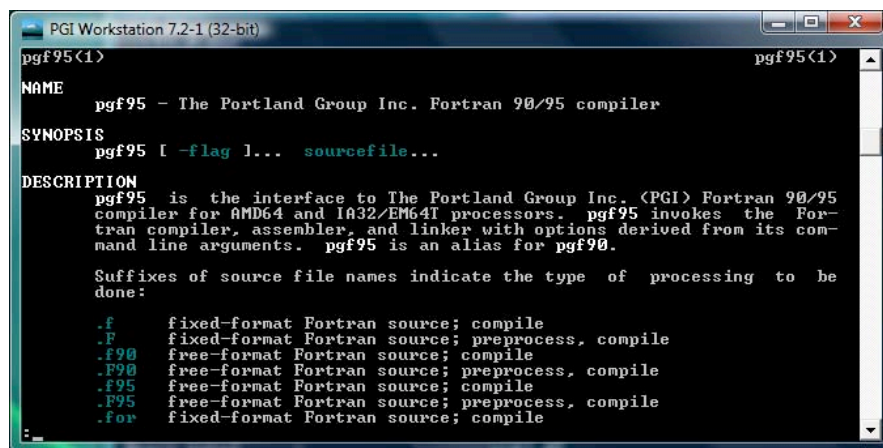


Figure 5: PGI Fortran (pgf95) UNIX man page

While the vi editor is included with the PGI Workstation distribution, many UNIX and Linux HPC developers prefer the EMACS editor and debugging environment. A pre-built binary

version is freely available for download and installation on Windows, and can be integrated for use with the PGDBG® debugger using the grand unified debugger (gud) interface.

Once you've installed PGI Workstation, a typical desktop working environment might look as follows:

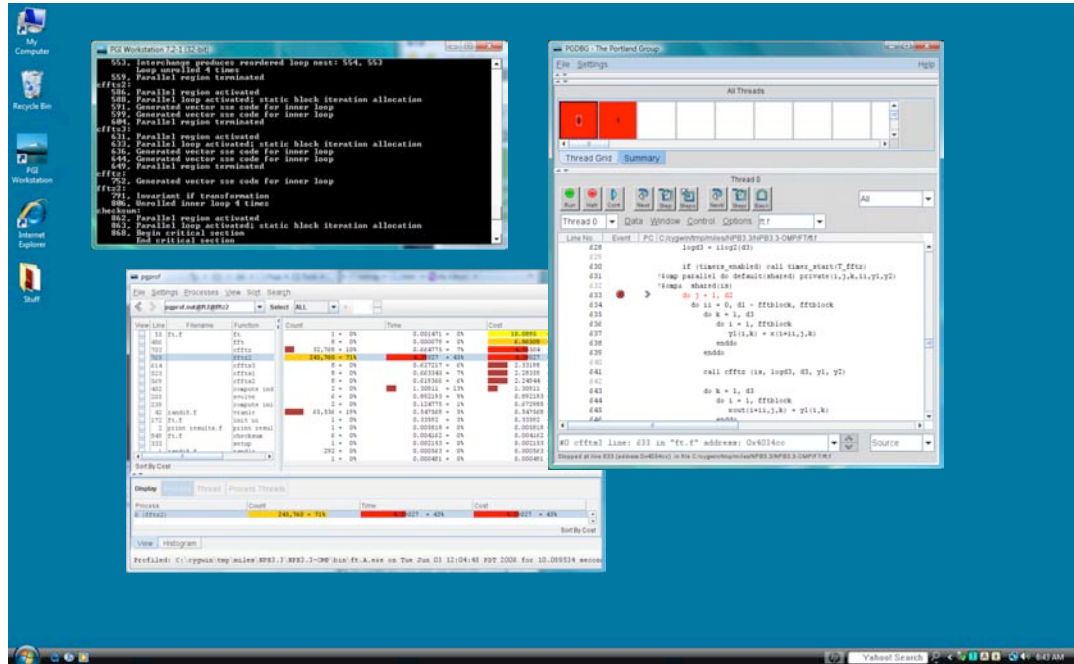


Figure 6: PGI Workstation desktop environment

With a classic UNIX-like development environment in place, we're ready to move on to porting and tuning OpenMP and MPI parallel applications in a familiar command-level HPC development environment.

Porting the OpenMP NAS Parallel Benchmarks to Windows

The OpenMP shared-memory parallel programming model is defined by a set of Fortran directives or C/C++ pragmas, library routines and environment variables that can be used to specify shared-memory parallelism on multi-core or mutli-processor systems. The directives and pragmas include a parallel region construct for writing coarse-grain parallel programs, work-sharing constructs which specify that a Fortran *DO* loop or C/C++ *for* loop iterations should be split among the available threads of execution, and synchronization constructs.

The OpenMP data environment is controlled either by using clauses on the directives or pragmas, or with additional directives or pragmas. Run-time library routines are provided to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region.

Microsoft Visual C++ and the PGI and Intel optimizing Fortran/C/C++ compilers all support the OpenMP 2.5 standard. Enabling OpenMP compilation is as simple as adding a compiler option to the command line or appropriate property page:

<i>Compiler</i>	<i>Compiler option to enable OpenMP programming</i>
Microsoft Visual C++	/openmp (or the Language property page)
PGI Fortran/C/C++	-mp
Intel Fortran/C/C++	/Qopenmp

Table 2: OpenMP compiler switches

The NAS Parallel Benchmarks (NPBs) are a useful vehicle for showing the OpenMP and parallel programming features available for Windows HPC Server 2008. They are a familiar set of benchmarks to many HPC users, are relatively small and manageable, and implement the same benchmark programs in serial form, as OpenMP shared-memory programs, and as MPI cluster programs. They also expose several of the minor working issues one encounters when porting classic HPC programs to Windows. These issues are very easy to overcome with a little guidance, as is shown in the tutorial that follows.

Getting Started

You can obtain an unmodified copy of the NAS Parallel Benchmarks by registering at <http://www.nas.nasa.gov/Resources/Software/npb.html>. For convenience, we've included a copy in the "Classic HPC" samples directory that is a companion to this white paper.

The NAS Parallel Benchmarks (NPB) are distributed in a form oriented to UNIX-like development environments. To get started, double-left-click on the PGI Workstation desktop icon to bring up a BASH command shell. Once the shell window is up, navigate to the working directory to see the unmodified NPB distribution file. We can use the gunzip and tar utilities to unpack it:

```

PGI$ ls
NPB3.3.tar.gz
PGI$ gunzip NPB3.3.tar.gz
PGI$ ls
NPB3.3.tar
PGI$ tar xvf NPB3.3.tar
NPB3.3/NPB3.3-MPI/
NPB3.3/NPB3.3-MPI/CG
. . .
NPB3.3/Changes.log
PGI$ ls
Changes.log      NPB3.3-JAV.README  NPB3.3-OMP  README
NPB3.3-HPF.README  NPB3.3-MPI        NPB3.3-SER

```

Several versions of the NAS Parallel benchmarks are included in this distribution, which was updated in August, 2007. Let's navigate down and take a look at the OpenMP versions of the benchmarks:

```

PGI$ cd NPB3.3/NPB3.3-OMP
PGI$ ls
BT  DC  FT  LU  Makefile  README.install  UA  common  sys
CG  EP  IS  MG  README    SP                bin  config
PGI$

```

Using Make to Build the FT Benchmark

The README.install file states that an individual benchmark is built using the command "make <benchmark> CLASS=<class>" where <benchmark> is one of (BT, SP, LU, FT, CG, MG, EP, UA, IS, DC) and <class> is one of (S, W, A, B, C, D, E). Let's try to build the Class A FT benchmark:

```

PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                   =
=      F77/C                             =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
Makefile:5: ../config/make.def: No such file or directory
make[1]: *** No rule to make target `../config/make.def'.  Stop.
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make: *** [ft] Error 2
PGI$

```

We need a well-defined config/make.def file to continue. Another look at the README.install indicates that several samples are available in the config/NAS.samples subdirectory, including one called make.def_pgi for the PGI compilers. Let's copy the pre-configured sample for the PGI compilers and try again:

```

PGI$ cp config/NAS.samples/make.def_pgi config/make.def
PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                  =
=      F77/C                            =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make[2]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
cc -o setparams setparams.c
make[2]: cc: Command not found
make[2]: *** [setparams] Error 127
make[2]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
make[1]: *** [config] Error 2
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make: *** [ft] Error 2

```

The makefiles are trying to use the command `cc` to compile C language files. It is very common for UNIX or Linux systems to be configured with a default C compiler available through the `cc` command, but we don't have such a default in our environment. We can use the `grep` utility to see where `cc` is used in the makefiles in our build directories:

```

PGI$ grep "\<cc" *ake* */*ake*
config/make.def:UCC      = cc
config/make.def.template:CC = cc
config/make.def.template:UCC      = cc
sys/Makefile:UCC = cc
PGI$

```

It occurs in only a few places to define a makefile variable. We can use the `vi` editor to edit `config/make.def` and `sys/makefile` to redefine `UCC` to `pgcc`, an ANSI C compiler available in our working environment. After doing so, we can try the build again:

```

PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                  =
=      F77/C                            =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make[2]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
pgcc -o setparams setparams.c
NOTE: your trial license will expire in 14 days, 9.59 hours.
make[2]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
../sys/setparams ft A
pgf77 -c -O3 -mp ft.f
cd ../common; pgf77 -c -O3 -mp randi8.f
cd ../common; pgf77 -c -O3 -mp print_results.f
cd ../common; pgf77 -c -O3 -mp timers.f
cd ../common; pgcc -c -O3 -o wtime.o ../common/wtime.c
PGC-F-0206-Can't find include file sys/time.h (../common/wtime.c: 4)
PGC/x86 Windows 7.2-1: compilation aborted
make[1]: *** [../common/wtime.o] Error 2
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make: *** [ft] Error 2
PGI$

```

We've made it a little further, but now there's a problem in the common/wtime.c source file. Let's take a look:

```

PGI$ cat common/wtime.c
#include "wtime.h"
#include <time.h>
#ifdef DOS
#include <sys/time.h>
#endif

void wtime(double *t)
{
    /* a generic timer */
    static int sec = -1;
    struct timeval tv;
    gettimeofday(&tv, (void *)0);
    if (sec < 0) sec = tv.tv_sec;
    *t = (tv.tv_sec - sec) + 1.0e-6*tv.tv_usec;
}
PGI$

```

The NPB authors tried to conditionally compile around the include of `<sys/time.h>` on Windows platforms, but the DOS pre-defined constant is not being recognized by `pgcc`. The constants `_WIN32` and `_WIN64` are much more standard for conditional compilation on Windows, and are recognized by almost any Windows compilers you might use. After editing the file and changing `DOS` to `_WIN64`, we try the build again:

```

PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                   =
=      F77/C                             =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make[2]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
../sys/setparams ft A
pgf77 -c -O3 -mp ft.f
cd ../common; pgf77 -c -O3 -mp randi8.f
cd ../common; pgf77 -c -O3 -mp print_results.f
cd ../common; pgf77 -c -O3 -mp timers.f
cd ../common; pgcc -c -O3 -o wtime.o ../common/wtime.c
PGC-S-0060-tv_sec is not a member of this struct or union (../common/wtime.c: 13)
PGC-S-0060-tv_sec is not a member of this struct or union (../common/wtime.c: 14)
PGC-S-0060-tv_usec is not a member of this struct or union (../common/wtime.c: 14)
PGC/x86 Windows 7.2-1: compilation completed with severe errors
make[1]: *** [../common/wtime.o] Error 2
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make: *** [ft] Error 2
PGI$

```

More difficulties compiling `wtime.c`. Let's search through the source files in the build directories to see where `wtime()` is used:

```

PGI$ grep wtime * */*.f
common/timers.f:c$      external          omp_get_wtime
common/timers.f:c$      double precision omp_get_wtime
common/timers.f:c$      t = omp_get_wtime()
common/timers.f:        call wtime(t)
PGI$

```

Tweaking the Timer Function

The only occurrence is in `common/timers.f`. If we open it up in the editor and take a look we see the following function near the bottom of the file:

```

        double precision function elapsed_time()
        implicit none
c$     external          omp_get_wtime
c$     double precision omp_get_wtime
        double precision t
        logical          mp
c ... Use the OpenMP timer if we can (via C$ conditional compilation)
        mp = .false.
c$     mp = .true.
c$     t = omp_get_wtime()
        if (.not.mp) then
c This function must measure wall clock time, not CPU time.
c Since there is no portable timer in Fortran (77)
c we call a routine compiled in C (though the C source may have
c to be tweaked).
            call wtime(t)
        endif
        elapsed_time = t
        return
        end

```

The timer function is set up to allow use of `omp_get_wtime()`, the OpenMP standard wall clock timer function. We can either replace the line

```
call wtime(t)
```

With a standard-conforming Fortran 90 intrinsic call to `system_clock()`:

```

        integer clock, hz
. . .
c     call wtime(t)
        call system_clock (count=clock,count_rate=hz)
        t = dble(clock) / dble(hz)
. . .

```

Or simply comment out the call to `wtime` in `timers.f`. When we try the build again:

```

PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                   =
=      F77/C                             =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make[2]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
../sys/setparams ft A
pgf77 -c -O3 -mp ft.fcd ../common; pgf77 -c -O3 -mp randi8.f
cd ../common; pgf77 -c -O3 -mp print_results.f
cd ../common; pgf77 -c -O3 -mp timers.f
cd ../common; pgcc -c -O3 -o wtime.o ../common/wtime.c
pgf77 -O3 -mp -o ../bin/ft.A ft.o ../common/randi8.o ../common/print_results.o .
../common/timers.o ../common/wtime.o
LINK : fatal error LNK1181: cannot open input file 'ft.o'
make[1]: *** [../bin/ft.A] Error 2
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make: *** [ft] Error 2
PGI$

```

All of the source files have compiled as expected, but the linker can't locate the ft.o object file.

Object File Filename Extensions

A look at the files in the FT directory reveals the problem:

```

PGI$ ls FT
Makefile  README  ft.f  ft.obj  global.h  inputft.data.sample  npbparams.h

```

On Windows, object files are generated by default with a .obj file extension. We could re-write the makefiles to account for this difference, but an easier solution is to explicitly name the object files during compilation. Let's search the makefiles for the -c compiler option, to see where it occurs:

```

PGI$ grep "\-c" *ake* */*ake*
sys/Makefile:FCOMPILE = $(F77) -c $(F_INC) $(FFLAGS)
sys/make.common:FCOMPILE = $(F77) -c $(F_INC) $(FFLAGS)
sys/make.common:CCOMPILE = $(CC) -c $(C_INC) $(CFLAGS)
PGI$

```

We can edit the definitions of FCOMPILE and CCOMPILE to explicitly name the output object file by using the automatic variable \$@, which expands to the filename of the target; after doing so, we can grep again to see the changes:

```

PGI$ !gr
grep "\-c" *ake* /*ake*
sys/Makefile:FCOMPILER = $(F77) -c $(F_INC) $(FFLAGS) -o $@
sys/make.common:FCOMPILER = $(F77) -c $(F_INC) $(FFLAGS) -o $@
sys/make.common:CCOMPILER = $(CC) -c $(C_INC) $(CFLAGS) -o $@
PGI$

```

When we try the build again:

```

PGI$ make FT CLASS=A
=====
=      NAS PARALLEL BENCHMARKS 3.3      =
=      OpenMP Versions                  =
=      F77/C                            =
=====

cd FT; make CLASS=A
make[1]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
make[2]: Entering directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/sys'
../sys/setparams ft A
pgf77 -c -O3 -mp -o ft.o ft.f
cd ../common; pgf77 -c -O3 -mp -o ../common/randi8.o randi8.f
cd ../common; pgf77 -c -O3 -mp -o ../common/print_results.o print_results.f
cd ../common; pgf77 -c -O3 -mp -o ../common/timers.o timers.f
pgf77 -O3 -mp -o ../bin/ft.A ft.o ../common/randi8.o ../common/print_results.o .
../common/timers.o ../common/wtime.o
make[1]: Leaving directory `/tmp/miles/NPB3.3/NPB3.3-OMP/FT'
PGI$

```

It looks like the whole program compiled and linked as expected, placing the resulting executable file in *bin/ft.A*.

A Successful Build of NAS FT

Let's see if we can execute it:

```

PGI$ bin/ft.A

NAS Parallel Benchmarks (NPB3.3-OMP) - FT Benchmark

Size           : 256x 256x 128
Iterations      :      6
Number of available threads :      1

T = 1      Checksum = 5.046735008193D+02      5.114047905510D+0
T = 2      Checksum = 5.059412319734D+02      5.098809666433D+02
T = 3      Checksum = 5.069376896287D+02      5.098144042213D+02
T = 4      Checksum = 5.077892868474D+02      5.101336130759D+02
T = 5      Checksum = 5.085233095391D+02      5.104914655194D+02
T = 6      Checksum = 5.091487099959D+02      5.107917842803D+02

Result verification successful
class = A

FT Benchmark Completed.
Class           = A
Size           = 256x 256x 128
Iterations      = 6
Time in seconds = 7.39
Total threads   = 1
Avail threads   = 1
Mop/s total     = 965.95
Mop/s/thread    = 965.95
Operation type  = floating point
Verification    = SUCCESSFUL
Version         = 3.3
Compile date    = 03 Jun 2008

Compile options:
  F77           = pgf77
  FLINK         = $(F77)
  F_LIB         = (none)
  F_INC        = (none)
  FFLAGS       = -O3 -mp
  FLINKFLAGS   = -O3 -mp
  RAND         = randi8

Please send all errors/feedbacks to:

NPB Development Team
npb@nas.nasa.gov

PGI$

```

Success. The program is running with one thread compiled using the `-O3` compiler optimization, and runs in 7.39 seconds. If we edit `config/make.def` and use the `-fast` option to enable SSE vectorization, the time improves to 7.13 seconds. In the case of the NAS FT benchmark, the more aggressive `-fast` compiler optimization improves performance only slightly. However, if you try it across all of the NAS Parallel Benchmarks you will find it makes a significant difference in several cases.

Note that the executable *ft.A* generated by the makefiles does not have a *.exe* extension, which is standard for Windows executable files. While it executed correctly in this case, it is advisable to change the makefiles to use executable filenames that are more standard—containing only one period and using the *.exe* extension. Editing the file *sys/make.common* and changing the first line from

```
PROGRAM = $(BINDIR)/$(BENCHMARK).$(CLASS)
```

to

```
PROGRAM = $(BINDIR)/$(BENCHMARK)_$(CLASS).exe
```

will avoid any potential confusion as to whether the files are executables. We assume this change has been made in the sections that follow.

Parallel Execution

As we see in the example above, the default for the PGI-compiled OpenMP executable is to use only 1 thread or process. This default is not standardized, and may vary depending on the compiler you are using. Some compilers use all available cores or CPUs by default. In any case, running OpenMP parallel executables on a specific number of cores or CPUs is simple.

Here's an example of using the standard `OMP_NUM_THREADS` environment variable to enable execution of the *ft_A.exe* on two cores of a dual-core system:

```

PGI$ export OMP_NUM_THREADS=2
PGI$ ./bin/ft_A.exe

NAS Parallel Benchmarks (NPB3.3-OMP) - FT Benchmark

Size                : 256x 256x 128
Iterations           :      6
Number of available threads :      2

T = 1      Checksum = 5.046735008193D+02    5.114047905510D+02
T = 2      Checksum = 5.059412319734D+02    5.098809666433D+02
T = 3      Checksum = 5.069376896287D+02    5.098144042213D+02
T = 4      Checksum = 5.077892868474D+02    5.101336130759D+02
T = 5      Checksum = 5.085233095391D+02    5.104914655194D+02
T = 6      Checksum = 5.091487099959D+02    5.107917842803D+02
Result verification successful
class = A

FT Benchmark Completed.
Class      = A
Size      = 256x 256x 128
Iterations = 6
Time in seconds = 4.67
Total threads = 2
Avail threads = 2
Mop/s total = 1528.80
Mop/s/thread = 764.40
. . .

PGI$

```

Performance improves to 4.67 seconds, a speed-up of 1.52 over the single-threaded version.

You can build and run all of the benchmarks using the *awk* script *config/suite.def*; you'll need to copy the *suite.def.template* file to *suite.def*, and make the appropriate edits in *suite.def* to specify the problem sizes. The entire NAS Parallel benchmarks suite can then be built using the command:

```
PGI$ make suite
```

In this case study, we've seen that using a UNIX-like command-level development environment—*BASH* shell, *vi* editor, *tar/gzip/make/grep/awk* utilities—it is straightforward to get the OpenMP NAS Parallel Benchmarks running using an optimizing Fortran compiler on Windows HPC Server 2008. The changes required were very simple, and included:

- Editing makefiles to remove the assumption that a C compiler *cc* is available by default
- Changing source files to use the standard `_WIN64` constant for conditional compilation
- Editing the *timer.f* file to enable use of an alternate timer function

- Editing the makefiles to force generation of object files with the `.o` extension, and executable files with a `.exe` extension

In general, application speed-up when running on multiple cores depends on the percentage of compute-intensive code you can parallelize, the efficiency with which it is parallelized, and working data set sizes. The overhead of starting up execution of a parallel loop or region is typically quite low—measured in hundreds of cycles for 2 cores and perhaps a few thousand cycles on 4 or 8 cores.

There are OpenMP standard environment variables that can be used to improve efficiency in some cases. `OMP_SCHEDULE` can be used to control the type of iteration scheduling to use for `DO` and `PARALLEL DO` loops that include the `SCHEDULE(RUNTIME)` clause. An important variable for mixed OpenMP and MPI programming is `OMP_WAIT_POLICY`, which controls the behavior of idle threads. It can be set to `ACTIVE` or `PASSIVE`. If set to `ACTIVE`, idle threads will spin on a semaphore consuming CPU time; if set to `PASSIVE`, idle threads will sleep and relinquish control of their core or CPU. Performance of hybrid MPI+OpenMP programs sometimes improves dramatically if the wait policy is set to `PASSIVE`, ensuring that the master thread performing MPI communications has sole access to shared compute node resources.

There are also vendor-specific environment variables which can be useful for ensuring maximum performance. The PGI compilers support the `MP_BIND` environment variable (set to `yes` or `y`) to enable binding of OpenMP threads to cores for the duration of a run. This ensures that threads do not hop from core to core during a run, which can degrade performance due to cache considerations and excessive context switching. The PGI compilers also support the `MP_BLIST` function which maps logical OpenMP threads to specific cores. For example “`export MP_BLIST=3,2,1,0`” will cause Core 3 to be mapped to OpenMP thread 0, etc.

Refer to both the OpenMP standard and the compiler-specific documentation for more details on how to use these and other environment variables and features to maximize multi-core performance.

Porting the MPI NAS Parallel Benchmarks to Windows

In this section we will get the MPI versions of the NAS Parallel Benchmarks running on a Windows HPC Server 2008 cluster using MS MPI. Most of the modifications required for the MPI benchmarks are identical to those in the OpenMP case study above; if any of the descriptions below seem too brief, refer to the OpenMP case study above for details. Again in this tutorial we can leverage the PGI Workstation UNIX-like shell environment to build and run the NPB MPI programs with very few modifications.

To get started, bring up a PGI Workstation *bash* command shell, navigate to the NPB3.3-MPI working directory, and create a *config/make.def* file from the template provided by NASA:

```
PGI$ cd ../NPB3.3-MPI
PGI$ cp config/make.def.template config/make.def
PGI$ vi config/make.def
```

Edit the *config/make.def* file and make the following changes:

- define MPIF77 = pgf95
- define MPICC = pgcc
- define CC = pgcc
- remove -lmpi from the definition of FMPI_LIB
- define FFLAGS and FLINKFLAGS = -fast -Mmpi=msmpi
- define CFLAGS and CLINKFLAGS = -fast -Mmpi=msmpi

The PGI command-level compiler drivers (*pgf95*, *pgcc*, *pgcpp*) all support the option **-Mmpi=msmpi** which automatically pulls in the appropriate header files, libraries and startup files to create an MS MPI application.

As in the OpenMP tutorial above, we need to edit the *sys/make.common* included makefile to explicitly name object files with a *.o* extension by adding a *-o \$@* to the end of the definitions of *FCOMPILE* and *CCOMPILE*.

Once that is complete, we can try to build the BT MPI benchmark:

```

PGI$ make BT CLASS=A NPROCS=1
=====
=      NAS Parallel Benchmarks 3.3      =
=      MPI/F77/C                        =
=====

cd BT; make NPROCS=1 CLASS=A SUBTYPE= VERSION=
make[1]: Entering directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
make[2]: Entering directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/sys'
pgcc -g -o setparams setparams.c
make[2]: Leaving directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/sys'
../sys/setparams bt 1 A
make[2]: Entering directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
pgf95 -c -I/usr/local/include -fastsse -Mmpi=msmpi -o bt.o bt.f
pgf95 -c -I/usr/local/include -fastsse -Mmpi=msmpi -o make_set.o make_set.f
pgf95 -c -I/usr/local/include -fastsse -Mmpi=msmpi -o initialize.o initialize.f

. . .

cd ../common; pgf95 -c -I/usr/local/include -fastsse -Mmpi=msmpi -o ../common/ti
mers.o timers.f
make[3]: Entering directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
pgf95 -c -I/usr/local/include -fastsse -Mmpi=msmpi -o btio.o btio.f
pgf95 -fastsse -Mmpi=msmpi -o ../bin/bt.A.1 bt.o make_set.o initialize.o exact_s
olution.o exact_rhs.o set_constants.o adi.o define.o copy_faces.o rhs.o solve_su
bs.o x_solve.o y_solve.o z_solve.o add.o error.o verify.o setup_mpi.o ../common/
print_results.o ../common/timers.o btio.o -L/usr/local/lib
make[3]: Leaving directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
make[2]: Leaving directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
make[1]: Leaving directory `/tmp/miles/work/NPB3.3/NPB3.3-MPI/BT'
PGI$

```

It seems to have built correctly. Let's see if we can execute it using the standard MPI 2 **mpiexec** command:

```

PGI$ mpiexec -n 1 ./bin/bt_A_1.exe

NAS Parallel Benchmarks 3.3 -- BT Benchmark

No input file inputbt.data. Using compiled defaults
Size: 64x 64x 64
Iterations: 200 dt: 0.0008000
Number of active processes: 1

Time step 1
Time step 20
Time step 40
Time step 60
Time step 80
Time step 100
Time step 120
Time step 140
Time step 160
Time step 180
Time step 200

Verification being performed for class A
accuracy setting for epsilon = 0.1000000000000E-07
Comparison of RMS-norms of residual
  1 0.1080634671464E+03 0.1080634671464E+03 0.8021787200786E-14
  2 0.1131973090122E+02 0.1131973090122E+02 0.1726182839850E-14
  3 0.2597435451158E+02 0.2597435451158E+02 0.1094221972590E-14
  4 0.2366562254468E+02 0.2366562254468E+02 0.9307519701617E-14
  5 0.2527896321175E+03 0.2527896321175E+03 0.1394160013545E-13
Comparison of RMS-norms of solution error
  1 0.4234841604053E+01 0.4234841604053E+01 0.0000000000000E+00
  2 0.4439028249700E+00 0.4439028249700E+00 0.0000000000000E+00
  3 0.9669248013635E+00 0.9669248013635E+00 0.3444599899784E-15
  4 0.8830206303977E+00 0.8830206303977E+00 0.2514602686293E-15
  5 0.9737990177083E+01 0.9737990177083E+01 0.3648302795747E-15
Verification Successful

BT Benchmark Completed.
Class = A
Size = 64x 64x 64
Iterations = 200
Time in seconds = 138.44
Total processes = 1
Compiled procs = 1
Mop/s total = 1215.58
Mop/s/process = 1215.58
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 31 May 2008

Compile options:
  MPIF77 = pgf95
  FLINK = $(MPIF77)
  FMPI_LIB = -L/usr/local/lib

```

```
FMPI_INC      = -I/usr/local/include
FFLAGS        = -fast -Mmpi=msmpi
FLINKFLAGS    = -fast -Mmpi=msmpi
RAND          = (none)
```

Please send the results of this run to:

NPB Development Team
Internet: npb@nas.nasa.gov

If email is not available, send this to:

MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035-1000

Fax: 650-604-3957

PGI\$

Success. Notice that we have to specify the number of MPI processes that will be used by the executable as part of the build step. This is not the case with all MPI programs, but is an artifact of the way the NAS Parallel Benchmarks are designed. As a result, we need to re-build BT in order to run using 4 MPI processes. Following is the (abbreviated) the output:

```
PGI$ make BT CLASS=A NPROCS=4
PGI$ mpiexec -n 4 ./bin/bt_A_4

NAS Parallel Benchmarks 3.3 -- BT Benchmark

No input file inputbt.data. Using compiled defaults
Size: 64x 64x 64
Iterations: 200 dt: 0.0008000
Number of active processes: 4

Time step 1
. . .
Verification Successful

BT Benchmark Completed.
Class = A
Size = 64x 64x 64
Iterations = 200
Time in seconds = 137.82
Total processes = 4
Compiled procs = 4
Mop/s total = 1221.04
Mop/s/process = 305.26

. . .
```

The BT benchmark seems to be running correctly, but is not showing any speed-up from 1 to 4 MPI processes. It runs in about 138 seconds in both cases. Why? By default, **mpiexec** will schedule all processes on the system where it is invoked. To run across multiple nodes of the

cluster, you need to ensure your working directories are shared across all cluster nodes, and execute using the job submission utility.

To see the available shared storage on your cluster, bring up the file. Shared folders will be denoted with a small “people” icon, and if you right-click on the folder, left click on properties, and open the sharing tab, you will see the network path of the shared storage. For example, the network path to a shared folder “Scratch” on the head node might be \\HN-SDK\Scratch, which we will use in the example below.

While we had been working in the /tmp/work directory, we can use the following command to copy our working directory into a shared folder:

```
PGI$ cp -r /tmp/work //HN-SDK/Scratch/work
PGI$ cd //HN-SDK/Scratch/work
PGI$ ls
NPB3.3  NPB3.3.tar
PGI$ cd NPB3.3
PGI$ ls
Changes.log          NPB3.3-JAV.README  NPB3.3-OMP  README
NPB3.3-HPF.README  NPB3.3-MPI          NPB3.3-SER
PGI$ cd NPB3.3-MPI
PGI$ ls
BT  DT  FT  LU  MPI_dummy  README          SP  common  sys
CG  EP  IS  MG  Makefile   README.install  bin  config
PGI$ cd bin
PGI$ ls
bt_A_1.exe  bt_A_4.exe  bt_A.dwf
PGI$ pwd
//HN-SDK/Scratch/work/NPB3.3/NPB3.3-MPI/bin
```

Here's an example of a job submission command that will cause *bt_A_4.exe* to run on 4 cluster nodes, placing the output in the file "bt_A_4.stdout":

```

PGI$ job submit /numcores:4 /workdir:\\\\HN-SDK\Scratch\work\NPB
3.3\NPB3.3-MPI\bin /stdout:bt_A_4.stdout mpiexec bt_A_4.exe
Job has been submitted. ID: 691.
PGI$ ls
bt_A_1.exe  bt_A_4.exe  bt_A_4.stdout  bt_A_dwf
PGI$ ls -lt
total 2680
-rwx----- 1 Administrators None 2420 May 31 11:27 bt_A_4.stdout
-rwx----- 1 Administrator None 651264 May 31 11:23 bt_A_4
-rwx----- 1 Administrator None 50176 May 31 11:23 bt_A_dwf
-rwx----- 1 Administrator None 652800 May 31 11:23 bt_A_1pwd
PGI$ cat bt_A_4.stdout

NAS Parallel Benchmarks 3.3 -- BT Benchmark

No input file inputbt.data. Using compiled defaults
Size: 64x 64x 64
Iterations: 200 dt: 0.0008000
Number of active processes: 4

Time step 1
. . .

BT Benchmark Completed.
Class = A
Size = 64x 64x 64
Iterations = 200
Time in seconds = 100.29
Total processes = 4
Compiled procs = 4
Mop/s total = 1677.94
Mop/s/process = 419.49
Operation type = floating point
Verification = SUCCESSFUL
Version = 3.3
Compile date = 31 May 2008

. . .

```

Notice that the time has improved from 138 seconds to 100 seconds. The Class A problem is quite small, which limits the opportunity for significant speed-up. Building the Class B or larger versions of the benchmarks will typically display much better speed-ups.

As with the OpenMP versions of the NAS Parallel Benchmarks, you can build and run all of the MPI benchmarks using the *awk* script *config/suite.def*; you'll need to copy the *suite.def.template* file to *suite.def*, and make the appropriate edits in *suite.def* to specify the problem sizes. The entire NAS Parallel benchmarks suite can then be built using the command:

```
PGI$ make suite
```

The Windows HPC Server 2008 Job Scheduler supports both command-line job submissions and an easy and intuitive graphical user interface. Refer to the white paper [Using Microsoft HPC Server 2008 Job Scheduler](#) for details on how to create, schedule, execute and monitor MSMPI jobs using the GUI.

Debugging MPI and OpenMP Programs

To demonstrate MPI and OpenMP debugging we will use the PGDBG debugger, which works on Windows HPC Server 2008 with nearly identical functionality to the version you may have used previously on Linux clusters. For source level debugging, you need to first compile your program with the `-g` option to generate debug information, such as program symbols and line number information. Below is a sample MPI program that we will use to demonstrate MPI debugging. The program distributes an array of 16 elements evenly across 4 nodes (using the `MPI_SCATTER` routine), and increments each element of the array in parallel:

```
program scatter
  include 'mpif.h'
  integer SIZE
  parameter(SIZE=4)
  integer numtasks, rank, sendcount, recvcount, source, ierr
  real*4 sendbuf(SIZE*SIZE), recvbuf(SIZE)
  data sendbuf /1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16/
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
  if (numtasks .eq. SIZE) then
    source = 1
    sendcount = SIZE
    recvcount = SIZE
    call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf, recvcount, MPI_REAL,
  source, MPI_COMM_WORLD, ierr)
    do i=1,SIZE
      recvbuf(i) = recvbuf(i) + 1
    enddo
    print *, 'rank= ',rank,' Results: ',recvbuf
  else
    print *, 'Program needs ',SIZE,' processors to run. Terminating.'
  endif
  call MPI_FINALIZE(ierr)
end program scatter
```

Using a text editor, enter the above program, and save it as **scatter.f90**. In a PGI Workstation command shell, compile the program with the following command:

```
PGI$ pgf90 -g -Mmpi=msmpi scatter.f90
```

Start up the debugger with the following command:

```
PGI$ pgdbg -c "file scatter.f90" -mpi -n 4 scatter.exe
```

We use the `-c "file scatter.f90"` option to specify a default file to display in the debugger. Without this option, no file is displayed because the debugger is initially stopped in a system library routine. The `-mpi` option takes a list of arguments we would normally pass to an `mpixec` command. So, `-n 4 scatter.exe` follows the `-mpi` option to specify 4

nodes and our executable, **scatter.exe**. For a complete list of options, enter **pgdbg -help** at the **PGI\$** prompt.

By default, PGDBG will invoke its Graphical User Interface (GUI), as shown below.

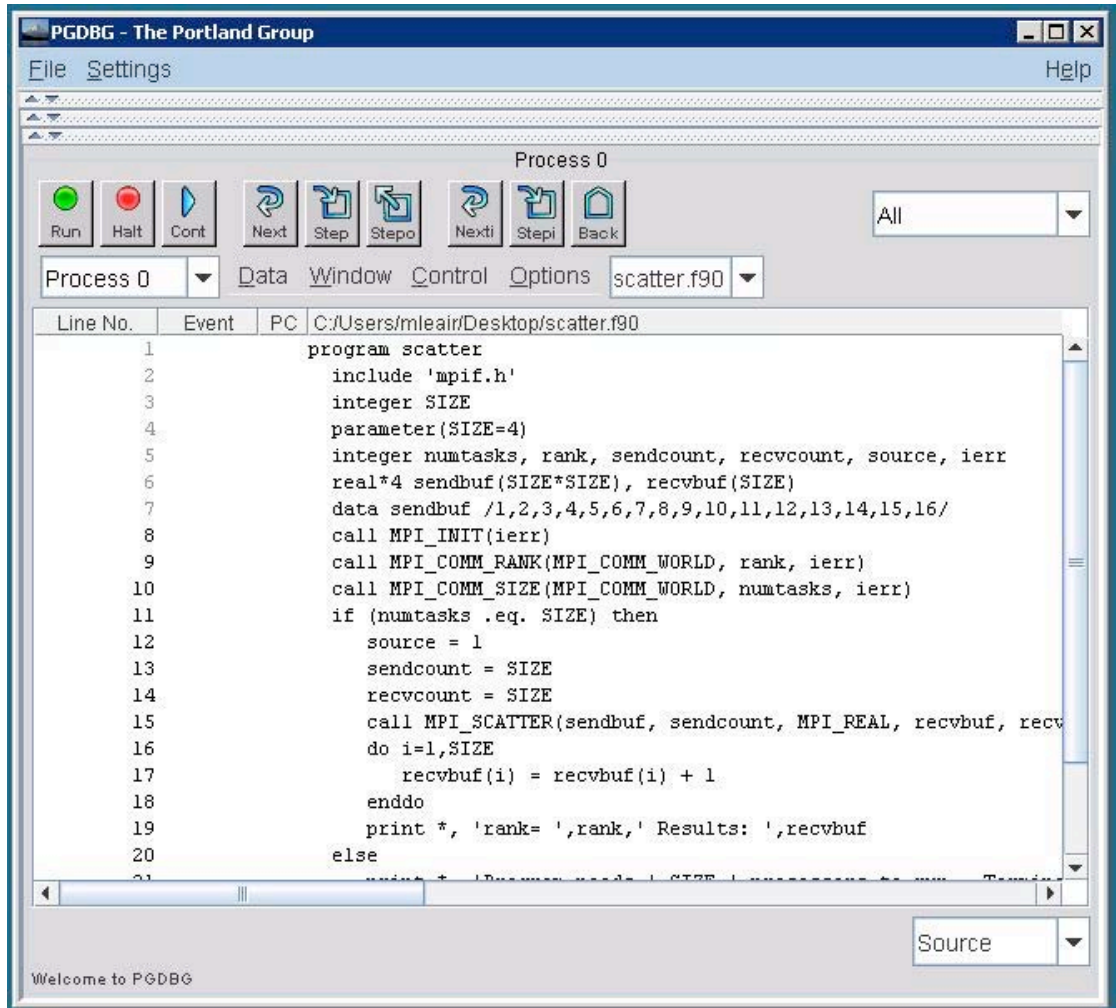


Figure 7: PGDBG graphical user interface

A full-featured command-line interface is also available if you specify the **-text** option. For example:

```
PGI$ pgdbg -text -c "file scatter.f90" -mpi -n 4 scatter.exe
```

To help show the status of each running process, the GUI can display a process grid. Drag the third from the top split pane (see figure below) to expose the process grid:

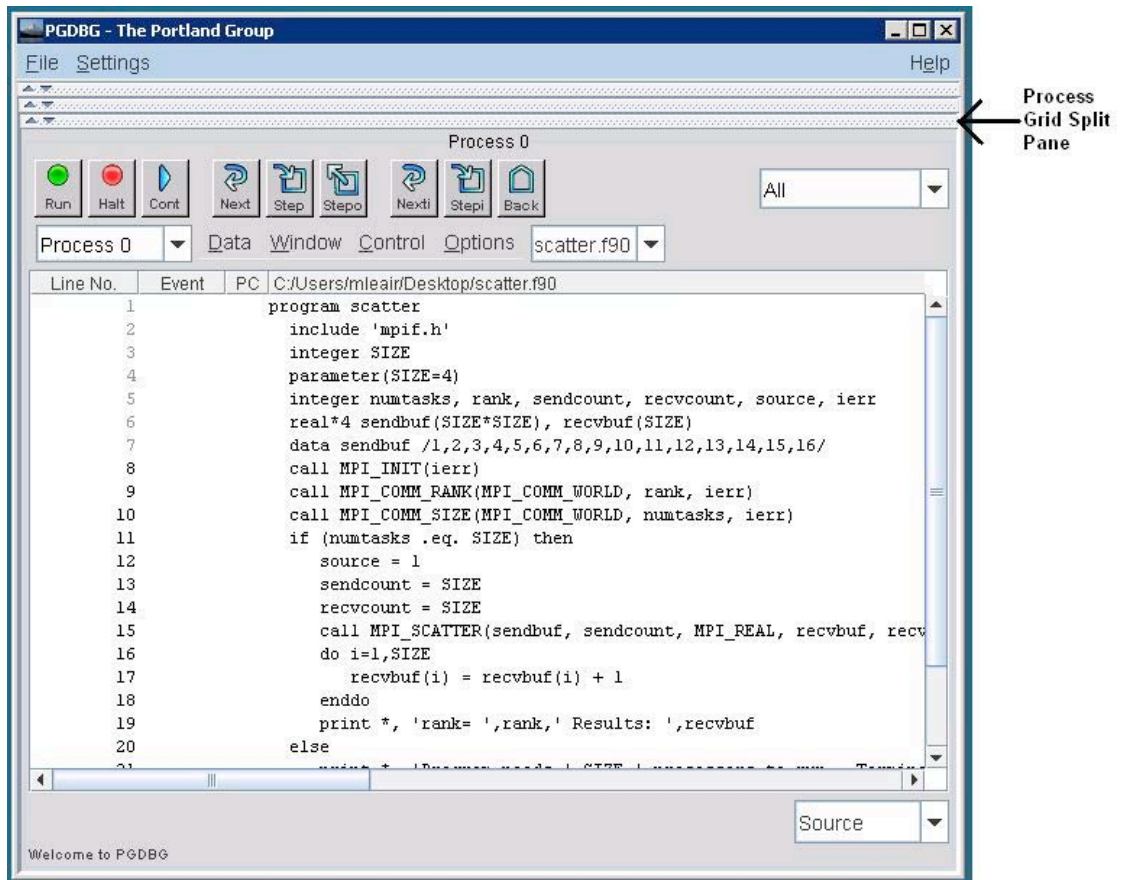


Figure 8: PGDBG process grid pane location

After exposing the process grid, you will probably want to resize the GUI to show an ample amount of source code. For example:

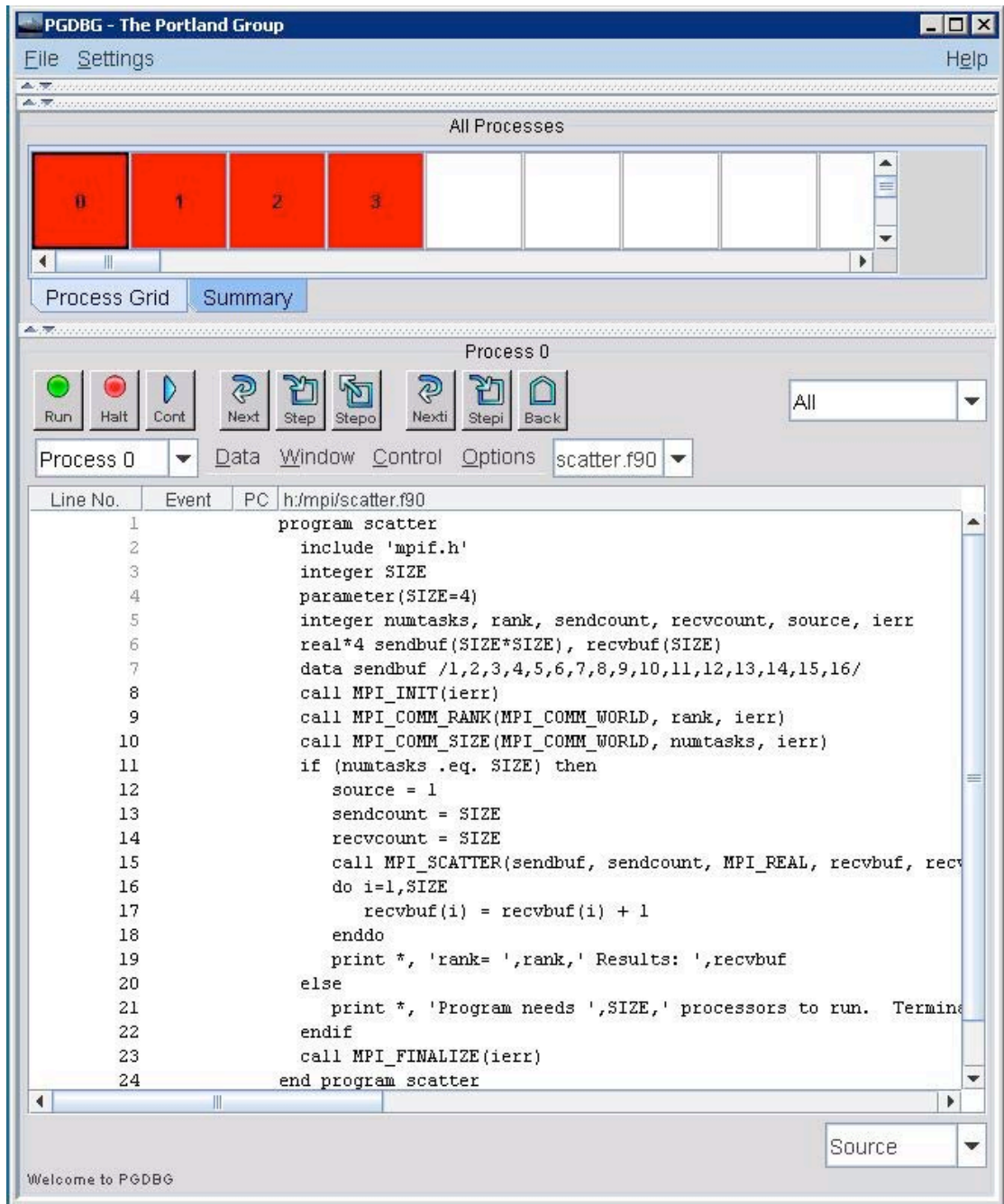
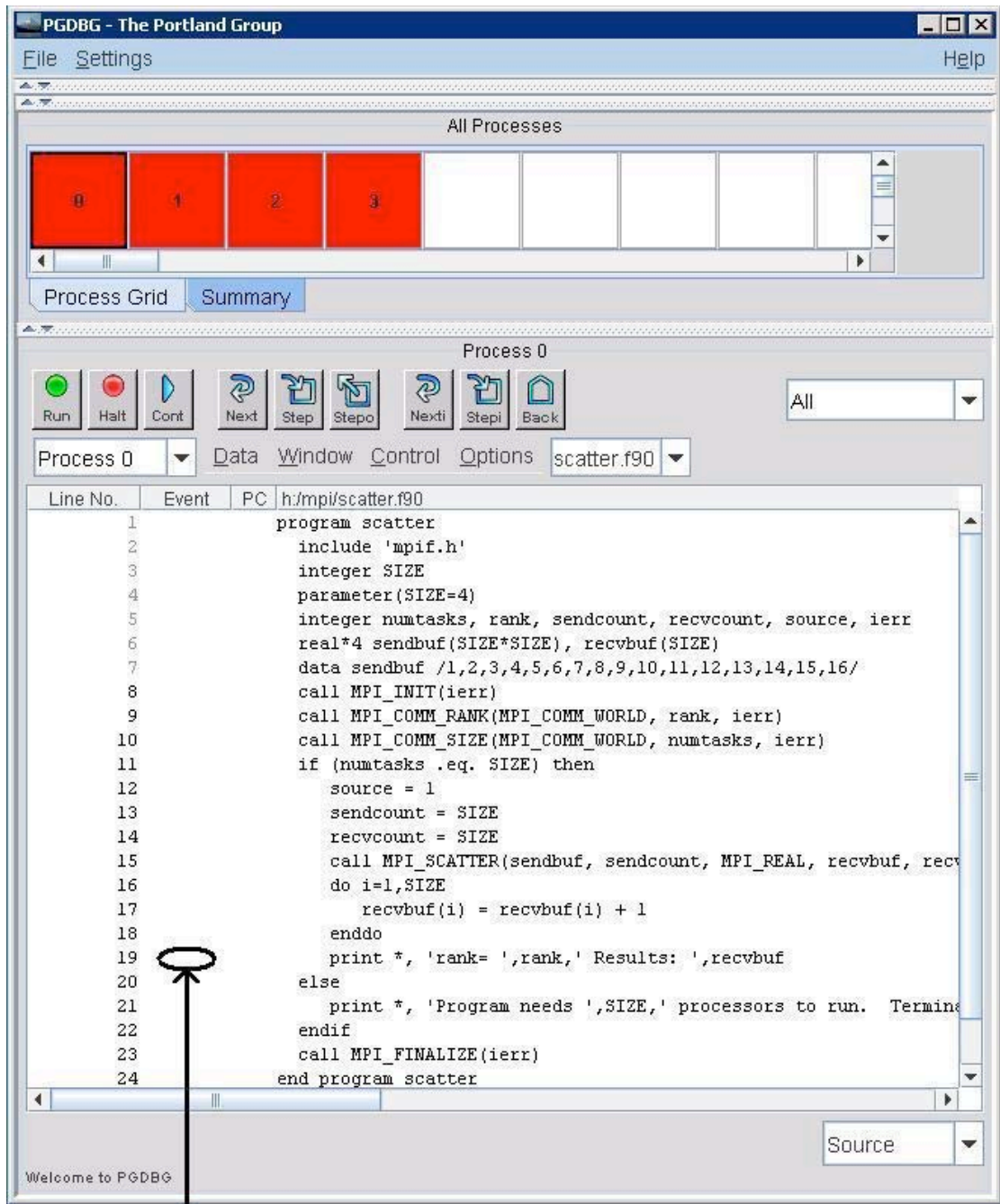


Figure 9: PGDBG process grid pane

The above figure shows 4 active MPI processes numbered 0,1, 2 and 3. The process grid reports a process' status through its color. When a grid element is red, the process associated with the grid element is in a stopped state. Green indicates that a process is running, blue indicates a process signaled, and black indicates the process was killed or exited.

To view data on each processor, we will view the **recvbuf** array on each processor by running our program to a breakpoint. You can set a breakpoint in the debugger by clicking on the desired line, under the Event column. For example, to set a breakpoint on line 19, click on the area to the right of 19, under the Event column (see figure below).



Click here to set a breakpoint on line 19

Figure 10: setting PGDBG breakpoints

A red “stop sign” icon is used to indicate a breakpoint in the graphical user interface. Click on the Cont button to continue execution of the program to the breakpoint set at line 19.

The figure below shows the debugger stopped on line 19 of the program.

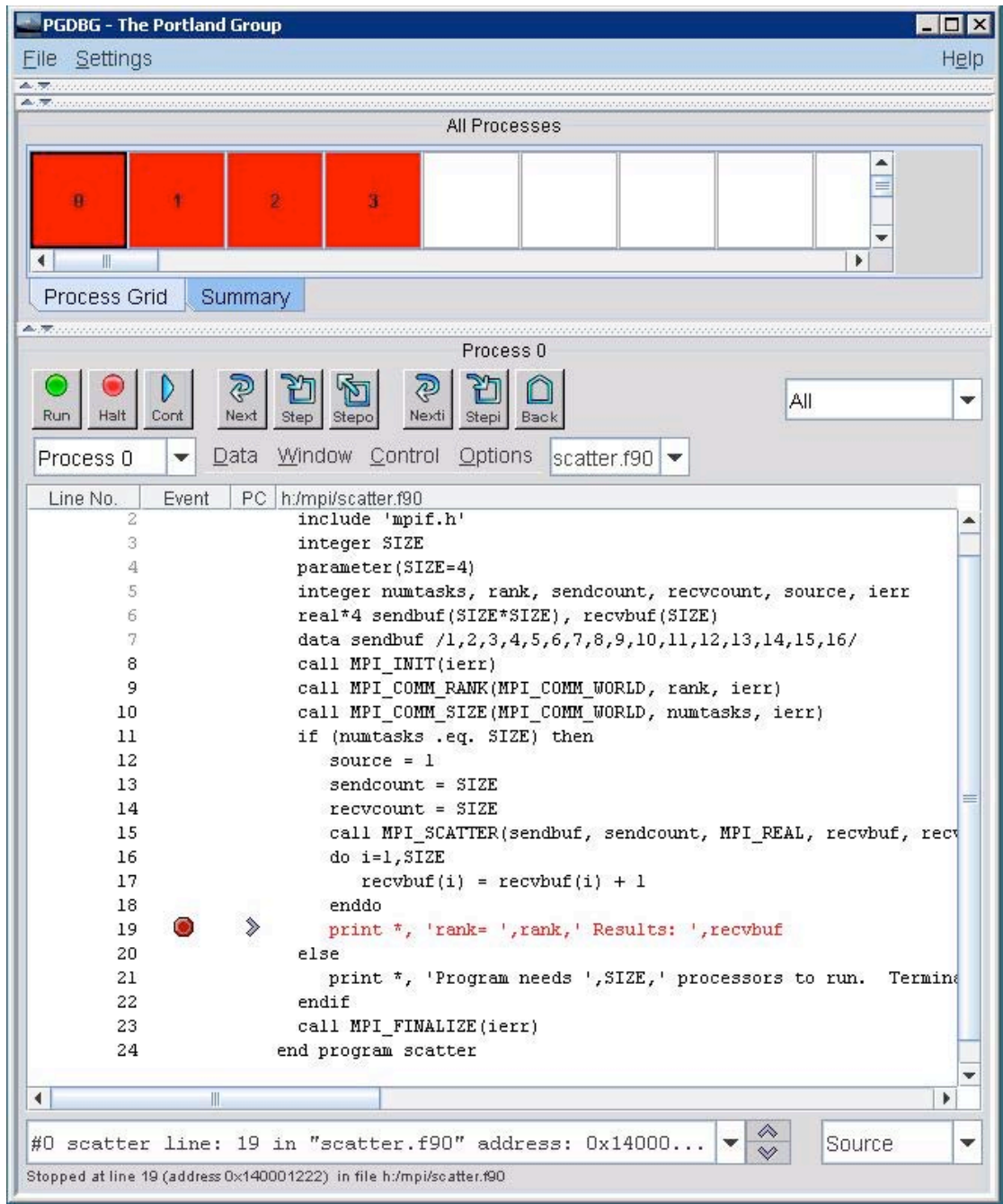


Figure 11: PGDBG breakpoint stop

The next line that will execute is indicated with the right arrow (➡). To print the **recvbuf** array for all processes, click on the Window menu, and select the Custom item, as shown below:

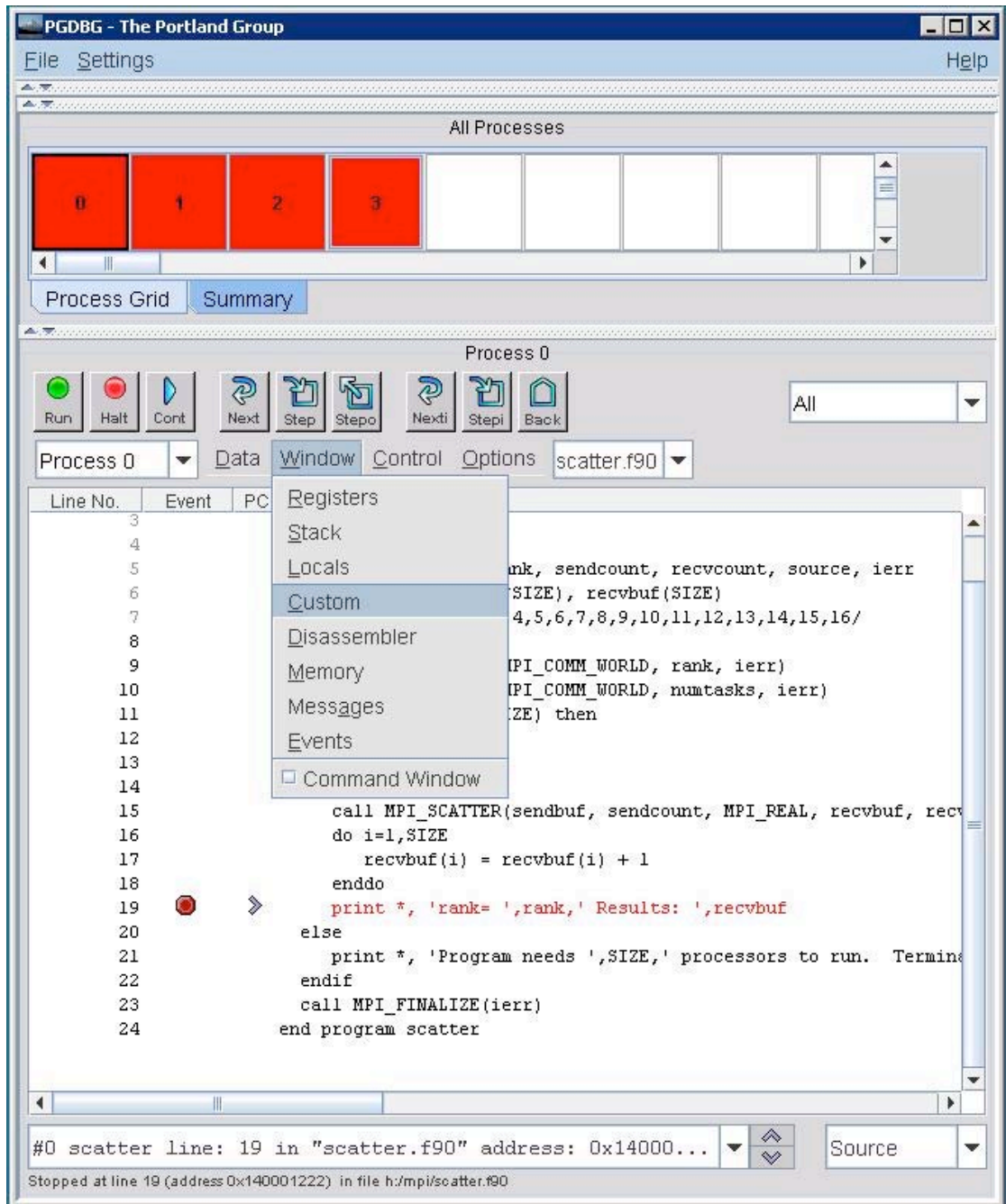


Figure 12: activate teh PGDBG custom window

The debugger will then display a custom window. In the **command>** text area, enter **print recvbuf**. Under the drop down box, change **Process 0** to **All Processes** (see figure below).

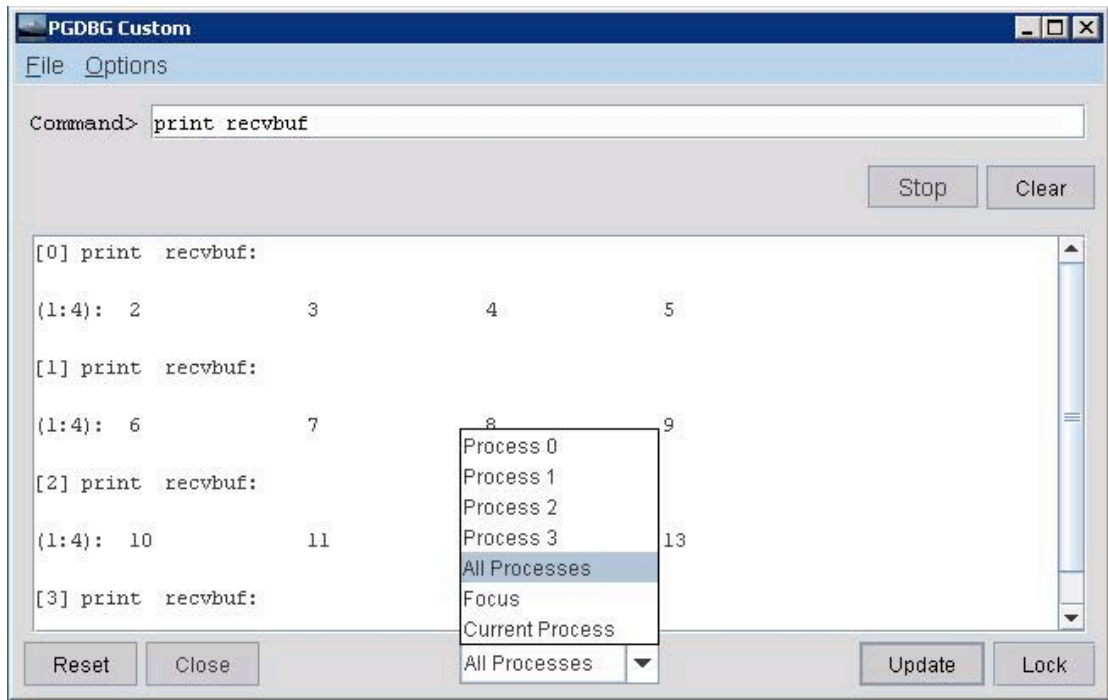


Figure 13: displaying process variables with PGDBG

You can now scroll the custom window to view the **recvbuf** array on each process. Each process has four array elements and both threads on each process have the same elements too. Below are the expected results in the custom window:

```

0] print  recvbuf:
(1:4):  2          3          4          5

[1] print  recvbuf:
(1:4):  6          7          8          9

[2] print  recvbuf:
(1:4): 10         11         12         13

[3] print  recvbuf:
(1:4): 14         15         16         17

```

After examining the **recvbuf** array, press the Cont button again to run the program to completion. After the program exits (all processes in the process grid will turn black), you can exit by selecting the Exit item under the File menu.

Debugging OpenMP programs is similar to debugging MPI programs. The PGDBG GUI operates essentially the same way for threads as it does for processes, and invoking the debugger for a parallel program is as simple as setting the environment variable OMP_NUM_THREADS and invoking PGDBG on the executable with no other arguments required.

If you are working over a slow remote connection, it can be impractical to use a graphical user interface. PGDBG includes a full-featured command line interface that can be used in these circumstances. We will use the following OpenMP sample code for this example:

```
        program omp_private_data
        integer omp_get_thread_num
        call omp_set_num_threads(2)
!$OMP PARALLEL PRIVATE(myid)
        myid = omp_get_thread_num()
        print *, 'myid = ',myid
!$OMP PARALLEL PRIVATE(nested_y)
        nested_y = 4 + myid
        print *, 'nested_y = ', nested_y
!$OMP END PARALLEL
!$OMP END PARALLEL
        end
```

The above program uses a private variable called **nested_y**. Because **nested_y** is private, each thread will get its own copy of **nested_y**. Using a text editor, enter the above program, and save it as **omp.f90**. In a PGI Workstation command shell, compile the program with the following command:

```
PGI$ pgf90 -mp -g omp.f90
```

Run the program to print the expected results for **nested_y**:

```
PGI$ omp.exe
myid =          0
  nested_y =          4
myid =          1
  nested_y =          5
```

The program output shows that the first thread (called thread 0) has a copy of **nested_y** equal to **4**. The second thread (called thread 1) has a copy of **nested_y** equal to **5**. Start up the debugger with the following command:

```
PGI$ pgdbg -text omp.exe
PGDBG 7.2-2 x86-64 (Cluster, 8 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc. All Rights Reserved.
Loaded: h:\omp\omp.exe

pgdbg>
```

Entering **help** after the **pgdbg>** prompt displays a list of commands:

```

pgdbg> help

PROCESS INIT
  args      attach  catch  debug  detach  ignore
  load      setargs

PROCESS CONTROL
  cont      halt     next   nexti   run     step
  stepi     stepout sync   synci   wait

PROCESS/THREAD SELECT
  proc      procs   thread threads

PROCESS/THREAD SETS
  p/t-set   focus   defset undefset viewset whichsets

EVENTS
  break     breaki  breaks delete  disable do      doi
  enable    hwatch  hwatchread hwatchboth status track tracki
  unbreak   unbreaki watch   watchi

PROGRAM LOCATIONS
  arrive    cd       disasm  edit   lines  list   pwd
  stacktrace stackdump where   /      ?

PRINTING
  ascii     bin     dec     display hex    print  printf
  oct       string undisplay

SYMBOLS AND EXPRESSIONS
  assign    call    decl    entry  lval   rval   set
  sizeof    type

SCOPE
  class     classes decls   down   enter  file   files
  global    names  scope  up     whereis

REGISTER ACCESS
  fp        pc      regs   retaddr sp

MEMORY ACCESS
  cread     dread   dump   fread  lread  iread  sread
  mqdump

CONVERSIONS
  addr      function line

MISCELLANEOUS
  alias     pgienv  directory help    history language log
  nop       repeat  script  setenv shell  sleep  unalias
  !         ^      quit

TARGET
  connect   disconnect native

pgdbg>

```

The **list** command, displays source lines:

```

pgdbg> list
#1:      program omp_private_data
#2:          integer omp_get_thread_num
#3:          call omp_set_num_threads(2)
#4:      !$OMP PARALLEL PRIVATE(myid)
#5:          myid = omp_get_thread_num()
#6:          print *, 'myid = ',myid
#7:      !$OMP PARALLEL PRIVATE(nested_y)
#8:          nested_y = 4 + myid
#9:          print *, 'nested_y = ', nested_y
#10:     !$OMP END PARALLEL

pgdbg>

```

Use the **break** command to set a breakpoint:

```

pgdbg> break 3
(1)breakpoint set at: omp_private_data line: "omp.f90"@3 address: 0x1400011411

```

The **run** command will run the program to the next breakpoint or to completion:

```

pgdbg> run
Loaded: C:/WINDOWS/system32/ntdll.dll
Loaded: C:/WINDOWS/system32/kernel32.dll
Breakpoint at 0x140001141, function omp_private_data, file omp.f90, line 3
#3:          call omp_set_num_threads(2)

pgdbg>

```

The **next** command will execute the next source line (line 3) and halt on the following source line (line 4):

```

pgdbg> next
([1] New Thread)
[0] Stopped at 0x140001155, function omp_private_data, file omp.f90, line 4
#4:      !$OMP PARALLEL PRIVATE(myid)

pgdbg [all] 0>

```

The call to **omp_set_num_threads(2)** creates two OpenMP threads. After executing this line, PGDBG prints **([1] New Thread)** to indicate a new thread was created. The **pgdbg** prompt also changed to indicate multi-thread mode. The **[all]** indicates that commands entered in the command prompt are applied to all threads and the **0** indicates the current thread is *thread 0* (the second thread is called *thread 1*).

The **print** command can be used to print data on the current thread. Let's run to line 9 and print the value of **nested_y** for each thread:

```

pgdbg [all] 0> break 9
[all] (2)breakpoint set at: omp_private_data line: "omp.f90"@9 address: 0x140001
281
2
pgdbg [all] 0> cont
[0] Breakpoint at 0x140001281, function omp_private_data, file omp.f90, line 9
#9:          print *, 'nested_y = ', nested_y

pgdbg [all] 0> list
#4:          !$OMP PARALLEL PRIVATE(myid)
#5:          myid = omp_get_thread_num()
#6:          print *, 'myid = ',myid
#7:          !$OMP PARALLEL PRIVATE(nested_y)
#8:          nested_y = 4 + myid
#9:==>>    print *, 'nested_y = ', nested_y
#10:         !$OMP END PARALLEL
#11:         !$OMP END PARALLEL
#12:         end
#13:

pgdbg [all] 0> print nested_y
4
pgdbg [all] 0> thread 1
[1] Breakpoint at 0x140001281, function omp_private_data, file omp.f90, line 9
#9:          print *, 'nested_y = ', nested_y
([1] Thread Stopped)

pgdbg [all] 1> print nested_y
5
pgdbg [all] 1>

```

Note that **nested_y** is **4** on thread 0 and **5** on thread 1. Also note the use of the `thread` command to switch the current thread from 0 to 1. The `threads` command can be used to view the status of all running threads:

```

pgdbg [all] 1> threads
0  ID  PID  STATE  SIG/CODE  LOCATION
   0   3464  Stopped  TRAP      omp_private_data line: "omp.f90"@9 addre
ss: 0x140001281
=> 1   3540  Stopped  TRAP      omp_private_data line: "omp.f90"@9 addre
ss: 0x140001281

pgdbg [all] 1>

```

Use the `cont` command to run the program to completion and the `quit` command to exit the debugger:

```
pgdbg [all] 1> cont
([*] Process Exited)

pgdbg [all]> quit
PGI$
```

In this case study, you were introduced to MPI and OpenMP debugging using the PGDBG GUI and Command Line interfaces. More information on PGDBG (including more detailed examples) can be found in the *PGI Tools Guide* available from The Portland Group website at <http://www.pgroup.com/doc/pgitools.pdf>.

Building the Linpack HPC Benchmark

HPL [1] is a benchmark used to measure the performance of Distributed Memory computer systems ranked in the Top 500 supercomputer list¹. The HPL benchmark solves a random dense linear system of equations using 64-bit double precision arithmetic. HPL is written in “C”, and uses MPI with either a BLAS [2] or a Vector Signal Processing Library (VSIPL).

Below are the steps for downloading, configuring, and running the HPL benchmark on Windows HPC Server 2008. This case study also uses a “C” compiler and a BLAS library found in the PGI Fortran, C and C++ compilers and tools for Windows. Another development environment can be used provided that it has access to a “C” compiler, BLAS math library, an MPI implementation, and GNU/Unix compatible unzip, tar, make, grep, and sort utilities.

Step 1: Download the HPL benchmark

Before beginning the benchmark, you will want to locate a disk partition that is shared on all of the nodes of your cluster. Download the benchmark into this shared partition. The benchmark is available from the following website: <http://www.netlib.org/benchmark/hpl> (click on the **Software** link to go to the download page). Download the benchmark (the file package is named **hpl.tgz**) into the shared partition. After you download the benchmark, you are ready to create a makefile. Follow the steps below to create the makefile.

Step 2: Create the Makefile

```
PGI$ gzip -dc hpl.tgz | tar xf -
PGI$ pwd
/d
PGI$ cd hpl/setup
PGI$ ./make_generic
PGI$ cd ..
PGI$ cp setup/Make.UNKNOWN Make.win64
```

After unpacking HPL with the **gzip** command, we executed the **pwd** command to show the path of our install directory. We will need this in a moment, so you may want to write it down or copy it into your computer’s clip board. In our example, the install directory is **/d**.

Step 3: Configure the Makefiles

Bring up the **Make.win64** makefile in an editor (vi, emacs, Notepad, etc):

¹ <http://www.top500.org>

```

#
# -----
# - Platform identifier -----
# -----
#
ARCH          = UNKNOWN
#
# -----
# - HPL Directory Structure / HPL library -----
# -----
#
TOPdir        = $(HOME)/hpl
INCdir        = $(TOPdir)/include
BINDir        = $(TOPdir)/bin/$(ARCH)
LIBdir        = $(TOPdir)/lib/$(ARCH)
#
HPLlib        = $(LIBdir)/libhpl.a
#

```

Change the line ARCH = UNKNOWN (below the Platform identifier line) to ARCH = WIN64.

Locate the TOPdir = \$(HOME)/hpl line and replace \$(HOME) with the install directory noted in Step 2. HPL needs a Windows pathname for this step, so you may have to convert this pathname to a Windows pathname. In step 2, the pwd command returned /d for the pathname. This is a Unix-like pathname. So, we will have to use the Windows pathname equivalent which is D:\.

```

#
# -----
# - Platform identifier -----
# -----
#
ARCH          = Win64
#
# -----
# - HPL Directory Structure / HPL library -----
# -----
#
TOPdir        = D:\hpl
INCdir        = $(TOPdir)/include
BINDir        = $(TOPdir)/bin/$(ARCH)
LIBdir        = $(TOPdir)/lib/$(ARCH)
#
HPLlib        = $(LIBdir)/libhpl.a
#

```

To configure the MPI libraries, locate the Mpinc = (located beneath the Message Passing library (MPI) line) and Mplib = lines. Change these lines to the pathname or compiler options for specifying the MPI include and library directories. If you are using the PGI compilers, simply specify -Mmpi=msmpi for both the Mpinc = and Mplib = lines, for example:

```

# -----
# - Message Passing library (MPI) -----
# -----
# Mpinc tells the C compiler where to find the Message Passing library
# header files, Mplib is defined to be the name of the library to be
# used. The variable Mkdir is only used for defining Mpinc and Mplib.
#
Mpdire      =
Mpinc       = -Mmpi=msmpi
Mplib      = -Mmpi=msmpi
#

```

To configure the compiler and linker options, locate the `CC = mpicc` (located beneath the Compilers / linkers—Optimization flags line) line in `Make.win64`. Set the `CC` and `LINKER` options to your compiler and linker. Set both of these variables to `pgcc` if you are using the PGI compilers and tools, for example:

```

# -----
# - Compilers / linkers - Optimization flags -----
# -----
#
CC          = pgcc
CCNOOPT    = $(HPL_DEFS)
CCFLAGS    = $(HPL_DEFS)
#
LINKER     = pgcc
LINKFLAGS  =
#
ARCHIVER   = ar
ARFLAGS    = r
RANLIB     = echo
#

```

We will use the CPU timing routine instead of the wall clock timing routine. To specify CPU time, we add `-DHPL_USE_CLOCK` to the `CCFLAGS =` line:

```

# -----
# - Compilers / linkers - Optimization flags -----
# -----
#
CC          = pgcc
CCNOOPT    = $(HPL_DEFS)
CCFLAGS    = -DHPL_USE_CLOCK $(HPL_DEFS)
#
LINKER     = pgcc
LINKFLAGS  =
#
ARCHIVER   = ar
ARFLAGS    = r
RANLIB     = echo
#

```

Save the modified Make.win64 file and close the editor.

```
PGI$ cd makes
```

Open the Make.ptest file with a text editor. Change the line

```
$(BINDir)/HPL.dat : ../HPL.dat
```

to

```
HPL.dat : ../HPL.dat.
```

The \$(BINDir) is removed for compatibility reasons. Next, remove \$(BINDir) in the \$(MAKE) \$(BINDir)/HPL.dat line as shown below:

```

## Targets #####
#
all      : dexe
#
dexe     : dexe.grd
#
HPL.dat  : ../HPL.dat
          ( $(CP) ../HPL.dat $(BINDir) )
#
dexe.grd: $(HPL_pteobj) $(HPLlib)
          $(LINKER) $(LINKFLAGS) -o $(xhpl) $(HPL_pteobj) $(HPL_LIBS)
          $(MAKE) HPL.dat
          $(TOUCH) dexe.grd
#

```

Save the Make.ptest file and close the editor.

Open the Make.timer file with a text editor. Remove the HPL_timer_walltime.o file located after HPL_timer_cputime.o and part of the HPL_timobj = assignment. We do not need this file since we are using CPU time instead of wall clock time.

```
## Object files #####  
#  
HPL_timobj      = \  
    HPL_timer.o      HPL_timer_cputime.o  
#
```

Save the Make.time file and close the editor.

```
PGI$ cd ..
```

Step 4: Build the Benchmark

```
make arch=win64
```

Below shows the results of a successful build. An unsuccessful build will display one or more error messages.

```
make[2]: Entering directory `/d/hpl/testing/ptest/win64'  
pgcc -o HPL_pddriver.o -c -DHPL_USE_CLOCK -DAdd_ -DF77_INTEGER=int -DStringSunStyl  
e -ID:\hpl/include -ID:\hpl/include/win64 -Mmpi=msmpi ../  
HPL_pddriver.c  
pgcc -o HPL_pdinfo.o -c -DHPL_USE_CLOCK -DAdd_ -DF77_INTEGER=int -DStringSunStyl  
e -ID:\hpl/include -ID:\hpl/include/win64 -Mmpi=msmpi ../HP  
L_pdinfo.c  
pgcc -o HPL_pdtest.o -c -DHPL_USE_CLOCK -DAdd_ -DF77_INTEGER=int -DStringSunStyl  
e -ID:\hpl/include -ID:\hpl/include/win64 -Mmpi=msmpi ../HP  
L_pdtest.c  
pgcc -o D:\hpl/bin/win64/xhpl HPL_pddriver.o      HPL_pdinfo.o  
      HPL_pdtest.o D:\hpl/lib/win64/libhpl.a -lblas -Mmpi=msmpi  
make HPL.dat  
make[3]: Entering directory `/d/hpl/testing/ptest/win64'  
( cp ../HPL.dat D:\hpl/bin/win64 )  
make[3]: Leaving directory `/d/hpl/testing/ptest/win64'  
touch dexe.grd  
make[2]: Leaving directory `/d/hpl/testing/ptest/win64'  
make[1]: Leaving directory `/d/hpl'  
PGI$
```

If you were successful in your build, then you are ready to execute a test run of the benchmark.

Step 5: Run the Benchmark

To run the benchmark, open a Windows Command Prompt. Next, use DOS commands to change the current drive and directory to the location of the HPL benchmark. For example,

```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

D:\>cd hpl

D:\hpl>cd bin\win64
```

We are now ready to run the benchmark with the HPC Job Manager (HJM) to schedule a run of HPL on your cluster. The **job** command takes several options; see **job -help** for a complete list. We use the **job submit** option to submit our job right away. The **/numcores** argument specifies the number of nodes to use. The **/stdout** argument specifies an output file. The **/stderr** argument specifies an error log. The **/workdir** argument is of the form **/workdir:\\HOST\DriveLetter\Pathname** where **HOST** is the name of the computer you are logged into, **DriveLetter** is the letter of the drive where **xhpl.exe** resides, and **Pathname** is the full path of the directory containing the **xhpl.exe** benchmark. For example,

```
D:\hpl\bin\win64>job submit /numcores:4 /stdout:HPL-default.out /std
err:HPL-default.err /workdir:\\%computername%\%CD:~0,1%\%CD:~2% mpiexec xhpl.exe

Job has been submitted. ID: 22.
D:\hpl\bin\win64>
```

In our example above, we use some DOS environment variables to automatically fill in **HOST**, **DriveLetter**, and **Pathname** in our **job submit** command (i.e., **%computername%** for **HOST**, **%CD:~0,1%** for **DriveLetter**, and **%CD~2%** for **PathName**

The status of your submitted job is displayed in the HJM Graphical User Interface (GUI). To view the status of your submitted job, start up the HJM from the Start Menu. After the GUI has loaded, find your job's ID in the top middle pane. Clicking on the job allows you to access more detailed information in the lower middle pane of the GUI.

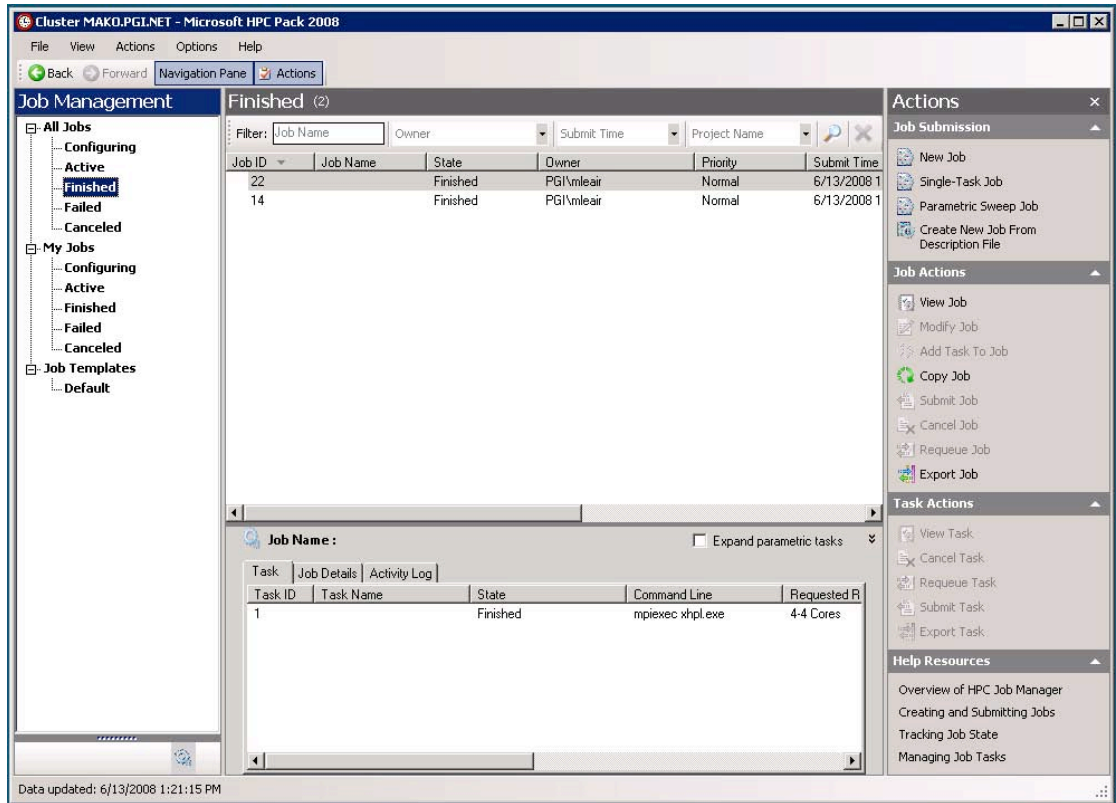


Figure 14: Windows HPC Server 2008 Job Manager

The HJM GUI's status column will indicate if your job is Queued, Running, Failed, or Finished. If the job runs successfully, as shown for job **22** above, the “Finished” state is indicated. If the job failed, you can click on the job and view any error messages in the lower window. You can also view the **HPL-default.err** file for any error information as well. If the program finished, you can view the results in the **HPL-default.out** file.

If the benchmark runs successfully, then you will see output similar to the following at the end of the output file (e.g., **HPL-default.out**) where the benchmark writes its output.

```
Finished      864 tests with the following results:
              864 tests completed and passed residual checks,
                0 tests completed and failed residual checks,
                0 tests skipped because of illegal input values.
```

Step 6: Review the Results

Below is a summary of pertinent information generated by the HPL benchmark and reported in the Top 500 List.

- **Rmax**—the performance in Gflop/s for the largest problem run on your system.
- **Nmax**—the size of the largest problem run on your system.
- **N1/2**—the size of the problem where half the Rmax execution rate is achieved.
- **Rpeak**—the theoretical peak performance Gflop/s for the machine.
- **#Proc.**—Number of processors used in the benchmark run.

Rmax can be found by locating the results for **Nmax** (the largest problem size specified in line 6 of HPL.dat). From that you can find **Rmax**, followed by **N1/2**.

By default, the benchmark is run 864 times with varying inputs. The top of the output file contains a description of the data presented for each run of the benchmark:

- T/V—Wall clock time / encoded variant.
- N —The order of the coefficient matrix A.
- NB—The partitioning blocking factor.
- P—The number of process rows.
- Q—The number of process columns.
- Time—Time in seconds to solve the linear system.
- Gflops—Rate of execution for solving the linear system.

Below is an example result in an **HPL-default.out** file:

```

=====
T/V              N    NB    P    Q              Time              Gflops
-----
WR00L2L2        29     1     2     2              0.01              1.513e-003
-----
||Ax-b||_oo / ( eps * ||A||_1 * N          ) =      0.0674622 ..... PASSED
||Ax-b||_oo / ( eps * ||A||_1 * ||x||_1   ) =      0.0519667 ..... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =      0.0174238 ..... PASSED

```

To find **Rpeak** in your output file, you need to locate the highest Gflop/s value. One way is to use the **grep** and **sort** commands available to a Unix shell, such as the BASH shell included with the PGI Workstation. Using the PGI Workstation command prompt, enter the following commands to get **Rpeak**:

```

PGI$ pwd
/d/hpl
PGI$ cd bin/win64
PGI$ grep 'WR' HPL-default.out | sort -o HPL-sort.out -g --key=7
PGI$

```

Open the **HPL-sort.out** file with a text editor and go to the last line in the file.

The **HPL-sort.out** file contains all 864 runs of the benchmark, but with the Gflop/s column sorted in ascended order. The last line of the output is **Rmax**. To illustrate, consider the last 14 lines of the following **HPL-sort.out** file:

WR00C2R4	34	4	1	4	0.00	1.823e-002
WR00R2C4	34	4	1	4	0.00	1.844e-002
WR00C2R2	35	4	1	4	0.00	1.857e-002
WR00R2R4	35	4	1	4	0.00	1.866e-002
WR00L2C2	35	4	1	4	0.00	1.869e-002
WR00R2L4	35	4	1	4	0.00	1.881e-002
WR00C2L4	35	4	1	4	0.00	1.905e-002
WR00R2R2	35	4	1	4	0.00	1.906e-002
WR00R2C4	35	4	1	4	0.00	1.909e-002
WR00R2L2	35	4	1	4	0.00	1.912e-002
WR00C2C4	35	4	1	4	0.00	1.922e-002
WR00C2L2	35	4	1	4	0.00	1.925e-002
WR00C2C2	35	4	1	4	0.00	1.953e-002
WR00L2L2	35	4	1	4	0.00	1.988e-002

The last column in the last line is **1.988e-002**, indicating **Rpeak** is **1.988e-002** Gflop/s.

To find **Nmax**, open the **HPL.dat** file in a text editor and locate the largest number in line 6. Below are the first 6 lines of the default **HPL.dat** file:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6           device out (6=stdout,7=stderr,file)
4           # of problems sizes (N)
29 30 34 35  Ns
```

According to the **HPL.dat** file above, **Nmax** is **35**. After finding **Nmax**, you can locate the best Gflop/s value for the **Nmax** problem size. In our simple example, **Rpeak** is the same as **Rmax**. However, in practice this is usually not the case since we will normally run larger problem sizes.

After finding **Rmax**, you are ready to find **N1/2**. In our example, half of **Rmax** is **9.994e-003** Gflop/s. In **HPL-sort.out**, you will want to find a Gflop/s value closest to half of **Rmax**. Using your text editor, navigate through the list of values until you find a Gflop/s value closest to your computed **Rmax/2**. Once you found **Rmax/2**, the value in the second column is **N1/2**.

Step 7: Tuning

Below are some of the ways to improve the performance of the HPL benchmark:

1. Tune the input data
2. Enable compiler optimizations
3. Use an improved BLAS library
4. Add more processors to your computer system

We will examine each of these in this section.

Tuning the Input Data

The HPL problem size is allowed to vary, and the best floating-point execution rate (measured in Gflop/s) is reported in the Top 500 List. Therefore, most tuning of the input data is related

to the problem size. Finding **Rpeak**, **Nmax**, **Rmax**, and **N1/2** will require some experimenting with the problem size specified in the **HPL.dat** file. Below are some of the common parameters you can customize for your HPL runs. For more information on tuning the HPL data, see the TUNING file located in HPL's top directory.

Line 5 in the **HPL.dat** file controls the number of different problem sizes that you wish to run. By default, the benchmark will run 4 different problem sizes. You can add up to 20 different problem sizes.

Line 6 in the **HPL.dat** file specifies the problem sizes. The default sizes are **29, 30, 34**, and **35**. You can change the problem sizes here and add more to the list. The higher the number, the more 64-bit floating point arithmetic (adds and multiplies in particular) gets executed by the benchmark.

Line 7 in **HPL.dat** specifies the number of block sizes (NBs). NB controls the amount of data distributed on each node. From a data distribution point of view, smaller the value for NB, better the load balance between processors. However, if you pick too small of a value for NB, performance may decrease due to an increase in the number of messages between processors.

Line 8 in **HPL.dat** specifies the different values for NB. The number of NBs should match the number specified in line 7.

Line 9 in **HPL.dat** specifies row major (default) or column major process mapping.

Line 10 in **HPL.dat** specifies the number of different matrix configurations. By default, the benchmark runs 3 different matrix configurations.

Line 11 in **HPL.dat** specifies the different sizes for the first dimension in the matrix.

Line 12 in **HPL.dat** specifies the different sizes for the second dimension in the matrix.

The default problem sizes (i.e., **29, 30, 34**, and **35**) are small and good for test runs. To really exercise the benchmark, larger problem sizes should be used. Let's rerun HPL using a larger problem size by modifying a couple of lines in the **HPL.dat** file. Replace **4** in line 5 with **1**, and replace **29 30 34 35** in line 6 with **2000**. Below shows the modified lines in **HPL.dat**:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6           device out (6=stdout,7=stderr,file)
1           # of problems sizes (N)
2000 Ns
```

Save the **HPL.dat** file and close the editor. We will also make a copy of this file since it gets reset each time you build the benchmark.

```
PGI$ cp HPL.dat HPL-new.dat
```

Rerun the benchmark in a Windows Command Prompt:

```
D:\hpl\bin\win64>job submit /numcores:4 /stdout:HPL-noopt.out /stderr:
HPL-noopt.err /workdir:\\%computername%\%CD:~0,1%\%CD:~2% mpiexec xhpl.exe
Job has been submitted. ID: 706.
D:\hpl\bin\win64>
```

Enable Compiler Optimizations

In our case study above, we did not enable any additional compiler optimizations. The purpose was to configure the benchmark and verify that it runs under the default configuration. After verifying that it builds and runs correctly, we can rebuild and run it using some compiler optimizations. Consult your compiler's documentation for a complete list of compiler optimizations. For the purposes of this discussion, we will use the **-fast** option and the **-Mipa=fast** compiler option available in the PGI compilers. The **-fast** option enables optimizations such as common subexpression elimination, loop unrolling, machine idiom recognition, SSE code generation, etc. The **-Mipa=fast** option performs interprocedural analysis such as constant propagation across function calls. We will also add a third option, **-Minfo=all**, to show what optimizations are applied to our benchmark.

Open the **Make.win64** file with a text editor and change the `CCFLAGS = -DHPL_USE_CLOCK $(HPL_DEFS)` to `CCFLAGS = -fast -Mipa=fast -Minfo=all -DHPL_USE_CLOCK $(HPL_DEFS)`. Next, change the `LINKFLAGS =` line to `LINKFLAGS = -fast -Mipa=fast`. For example,

```
#
# -----
# - Compilers / linkers - Optimization flags -----
# -----
#
CC          = pgcc
CCNOOPT     = $(HPL_DEFS)
CCFLAGS     = -fast Mipa=fast Minfo=all -DHPL_USE_CLOCK $(HPL_DEFS)
#
LINKER      = pgcc
LINKFLAGS   = -fast -Mipa=fast
#
ARCHIVER    = ar
ARFLAGS     = r
RANLIB      = echo
#
```

Save the **Make.win64** file and close your editor.

```
PGI$ make clean arch=win64
PGI$ make arch=win64 >& build-log
```

After building HPL, open up the **build-log** file with a text editor. Search for the first compile line in the file (i.e., search for **HPL_dlacpy.o**). Notice that the compilation includes *Loop unrolled* messages:

```

pgcc -o HPL_dlacpy.o -c -fast -Mipa=fast -Minfo=all -DHPL_USE_CLOCK -DAdd_ -DF77
_INTEGER=int -DStringSunStyle -ID:\hpl/include -ID:\hpl/include
/win64 -Mmpi=msmpi ../HPL_dlacpy.c
HPL_dlacpy:
  169, Loop unrolled 2 times
  289, Loop unrolled 8 times
  313, Loop unrolled 4 times
  337, Loop unrolled 8 times

```

Next, search for the **HPL_dlange.o** file. The compiler also generated vector SSE code for inner loops and performed some machine idiom recognition:

```

pgcc -o HPL_dlange.o -c -fast -Mipa=fast -Minfo=all -DHPL_USE_CLOCK -DAdd_ -DF77
_INTEGER=int -DStringSunStyle -ID:\hpl/include -ID:\hpl/include/win64 -Mmpi=msmpi
../HPL_dlange.c
HPL_dlange:
  145, Generated 2 alternate loops for the inner loop
      Generated vector sse code for inner loop
      Generated 1 prefetch instructions for this loop
      Generated vector sse code for inner loop
      Generated 1 prefetch instructions for this loop
      Generated vector sse code for inner loop
      Generated 1 prefetch instructions for this loop
  165, Memory zero idiom, loop replaced by call to __c_mzero8
  169, Loop unrolled 2 times

```

After rebuilding the benchmark, you can rerun and compare your results with the previous run. Before rerunning the benchmark, let's set the same input parameters as the previous run by overwriting **HPL.dat** with **HPL-new.dat**:

```

PGI$ cd bin/win64
PGI$ cp HPL-new.dat HPL.dat

```

Rerun the benchmark in a Windows Command Prompt:

```

D:\pl\bin\win64>job submit /numcores:4 /stdout:HPL-opt.out /stderr:
HPL-opt.err /workdir:\\%computername%\%CD:~0,1%\%CD:~2% mpiexec xhpl.exe
Job has been submitted. ID: 707.
D:\hpl\bin\win64>

```

After the benchmark runs, compare the results in **HPL-opt.out** with the results in **HPL-noopt.out**. The **Rpeak** in **HPL-out.out** should be higher than the **Rpeak** in **HPL-noopt.out** (locating **Rpeak** in the HPL output files was discussed in Step 6 of the previous section).

An Improved BLAS

Most of the execution of the HPL benchmark centers around the library routines that manipulate the matrices. By default, the BLAS library is used. On some computer architectures there are BLAS libraries that are tuned specifically for the hardware. Such a

library for the AMD Opteron processor is known as the AMD Core Math Library (ACML)². To use a tuned library, such as the ACML, change the **LAlib = -lblas** line in **Make.win64** to the desired math library and options. If you wish to use the ACML, then change this line to **LAlib = /FORCE:UNRESOLVED -lacml** For example,

```
#
# -----
# - Linear Algebra library (BLAS or VSIBL) -----
# -----
# LAinc tells the C compiler where to find the Linear Algebra library
# header files, LAlib is defined to be the name of the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir      =
LAinc      =
LAlib      = /FORCE:UNRESOLVED -lacml
#
```

After saving the modified Make.win64 file, you are ready to rebuild the benchmark:

```
PGI$ make arch=win64 clean
PGI$ make arch=win64
```

After the benchmark builds, you will want to go back into **bin/win64** directory and reset the **HPL.dat** file:

```
PGI$ cd bin/win64
PGI$ cp HPL-new.dat HPL.dat
```

Rerun the benchmark in a Windows Command Prompt:

```
D:\hpl\bin\win64> job submit /numcores:4 /stdout:HPL-acml.out /stderr:HPL-acml.err
/workdir:\\%computename%\%CD:~0,1%\%CD:~2% mpiexec xhpl.exe
Job has been submitted. ID: 708.
D:\hpl\bin\win64>
```

After the benchmark runs, compare the results in **HPL-acml.out** with the results in **HPL-opt.out**. The **Rpeak** in **HPL-acml.out** should be higher than the **Rpeak** in **HPL-opt.out**.

Increase Number of Processors

If you have the luxury of adding more nodes to your cluster, then you may be able to increase the performance of your parallel applications. It depends on how well your application scales. The HPL benchmark is highly parallel and will typically scale when you double the number of processors that participate in the benchmark's computation [3].

To rerun the benchmark with more nodes (assuming your cluster has 8 or more nodes):

² Available from <http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx>.

```
D:\hpl\bin\win64> job submit /numcores:8 /stdout:HPL-eight.out /stderr:HPL-eight.err  
/workdir:\\%computername%\%CD:~0,1%\%CD:~2% mpiexec xhpl.exe
```

After the benchmark runs, compare the results in **HPL-eight.out** with the results in **HPL-acml.out**. The **Rmax** in **HPL-eight.out** should be approximately two times as fast as the **Rmax** in **HPL-acml.out**.

After tuning HPL and computing **Rmax**, **Nmax**, **N1/2**, and **Rpeak** for your system, you are ready to compare your results with the Top 500 list for possible submission. To view the top 500 list, go to <http://www.top500.org>. On the website, mouse over the “Lists” tool bar item and select the most current date for the most current list. If your performance falls somewhere on the top 500, then you can submit your results by clicking on the “Submissions” tool bar item on the website.

Conclusion

While performance of individual x64 processor cores is still improving, a premium on power efficiency has led processor vendors to push aggressively on multi-core technology rather than increased clock speeds. Significant HPC application performance gains in the next few years will depend directly on the ability to exploit multi-core and cluster platforms.

There are now a variety of traditional HPC compilers and development tools available on Windows HPC Server 2008 that provide the ability to migrate incrementally from serial to auto-parallel or OpenMP parallel algorithms for multi-core processors. If you have a mature MPI application, or are ready to take the next step to cluster-enabled Windows applications using MSMPI, parallel debugging tools and math libraries are available to make porting and tuning of applications to MPI more tractable either in a traditional HPC development environment or using Microsoft Visual Studio 2008.

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 STMicroelectronics company.

All other marks are the property of their respective owners.

0908

References

- [1] Dongarra, J. J., Luszczek, P., and Petitet, A. 2003. The LINPACK Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience* 15, 1--18.
- [2] Dongarra, J. J., J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1-17, March 1990.
- [3] Antoine Petitet, R. Clint Whaley, Jack J. Dongarra, and Andy Clearly. *HPL—A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. Innovative Computer Laboratory, September 2000. Available at <http://www.netlib.org/benchmark/hpl/>.