Optimizing Application Performance on Cray Systems with PGI Compilers and Tools

Douglas Miles, Brent Leback *and* **David Norton**
The Portland Group (PGI)

ABSTRACT: *PGI Fortran, C and C++ compilers and tools are available on most Cray XT3 and Cray XD1 systems. Optimizing performance of the AMD Opteron processors in these systems often depends on maximizing SSE vectorization, ensuring alignment of vectors, and minimizing the number of cycles the processors are stalled waiting on data from main memory. The PGI compilers support a number of directives and options that allow the programmer to control and guide optimizations including vectorization, parallelization, function inlining, memory prefetching, interprocedural optimization, and others. In this paper we provide detailed examples of the use of several of these features as a means for extracting maximum single-node performance from Cray systems using PGI compilers and tools.*

KEYWORDS: Compiler, Fortran, C, C++, XT3, XD1, AMD Opteron, Optimization, Vectorization, Parallelization

1. Introduction

The Portland Group, Inc (PGI) develops and supports Fortran, C and C++ compilers for AMD Opteron (AMD64) processor-based CRAY XT3 and CRAY XD1 systems. These compilers generate highly optimized code for AMD64 processors, and are interoperable with the TotalView debugger and CrayPat performance analysis tools from Cray. We focus here on the optimization capabilities of the PGI compilers, and in particular on options and directives available to improve performance of PGI-compiled applications.

The PGI compilers implement several high-level optimizations including loop unrolling, loop vectorization, data prefetching, alternate code generation, interprocedural analysis and optimization, function inlining, loop parallelization and profile-feedback. In general, optimization involves using transformations and code replacements that are independent of the particular target, as well as transformations that take advantage of the AMD64 architecture, instruction set and registers.

Examples of target-independent transformations include loop interchange to maximize stride 1 accesses, loop unrolling to mitigate the cost of branch penalties, interprocedural constant propagation to enable various other optimizations, and function inlining to minimize overhead inherent in procedure calls. Examples of target-dependent transformations include vectorization to enable generation of AMD64 packed streaming SIMD extensions (SSE) instructions, data prefetching to mitigate the effects of memory latency, and instruction selection and scheduling to maximize efficiency of the AMD64 microarchitecture.

In the sections that follow, we will discuss the various categories of optimizations, and how the application programmer can affect or improve how and when these optimizations are performed.

2. SSE Vectorization

Vectorization is perhaps the single most important compiler optimization for AMD64 processors. It is used to identify and transform loops to take advantage of packed SSE instructions, which process 128 bits of data per instruction – either two 64-bit operands or four 32-bit operands. SSE vectorization can result in speed-ups of over a factor of two on loops that operate on 32-bit data, and large percentage speed-ups on loops that operate on

64-bit data. For floating-point computationally intensive loop-based applications, it is not uncommon to see 20 % - 30% speed-ups of the entire application on an AMD64 processor if most or all of the time-consuming loops can be vectorized.

To enable vectorization with any of the PGI compilers, use the **–fastsse** compiler option:

```
% pgf95 -fastsse -Minfo vadd.f
loop:
 18, Generating vector sse code for inner loop
     Generated 3 prefetch instructions for
        this loop
```

Note the use of **–Minfo**, which causes the compiler to emit messages detailing how loops are optimized. The name of the function being compiled is listed followed by a colon. The line numbers of optimized loops and any relevant messages are emitted slightly indented under the function name. This makes it easy to correlate optimizations with the loops in your original Fortran, C or C++ source code. During the the optimization phase, it is a good idea to use –Minfo for all of your compilations.

If you identify a time-intensive loop in your application, examine the –Minfo output to determine if the loop is vectorized. If not, try using –Mneginfo. Consider the following C code fragment:

```
void func4(float *u1, float *u2, float *u3, ...
...
for (i=-NE+1,p1=u2-ny,p2=u2+ny; i<nx+NE-1; i++)
    u3[i] += c1z*(p1[i] + p2[i]);
for (i=-NE+1; i<nx+NE-1; i++) {
    float vdt = v[i]*dt;
    u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
}
```

Compiled with pgcc –fastsse –Minfo –Mneginfo, the following messages are emitted:

```
221, Loop unrolled 4 times
221, Loop not vectorized due to data dependency
223, Loop not vectorized due to data dependency
```

The first loop at line 221 is unrolled but not vectorized. The second loop is not optimized at all. Why not? The compiler cannot determine that the memory regions pointed to by `u1`, `u2` and `u3` do not overlap. However, the programmer may know that these pointer-based arrays do not overlap. If so, the **–Msafeptr** option can provide a hint to the compiler to disambiguate the pointers passed in as arguments:

```
221, Generated vector sse code for inner loop
     Generated 3 prefetch instructions for
        this loop
223, Unrolled inner loop 4 times
```

The first loop is now vectorized. Note that –Msafeptr specifies all pointer-based arrays in a file are safe for optimization. See the *PGI User's Guide* for a detailed description of how to restrict optimization to certain types of pointers, or to give hints to the compiler at the loop level by inserting `safeptr` pragmas into your source code. In practice, pointers which cannot be statically disambiguated by a compiler are the single most limiting factor in vectorization of C and C++ applications.

Note that the second loop is still not vectorized. In the ANSI C language, by default all constants like the `2.` in the second loop, are default `double`. The mixed precision computations in this loop prevent vectorization. If the loop is re-compiled adding the **–Mfcon** option, which instructs the compiler to treat constants as default `float`, we see the following:

```
221, Generated vector sse code for inner loop
     Generated 3 prefetch instructions for this
        loop
223, Generated vector sse code for inner loop
     Generated 4 prefetch instructions for
        this loop
```

Both loops are now vectorized. This example is based on a real end-user code fragment sent in with a technical support inquiry to the PGI applications group. In this case, vectorization resulted in a speed-up of nearly 3X for the most compute-intensive function in the user's application.

There are many common barriers to vectorization. Loops with function calls are not vectorized; try **–Mipa=inline** or **–Minline** to inline the function. Loops with a large number of statements might fail the internal heuristic for profitability of vectorization; try overriding the heuristic using **–Mvect=nosizelimit** (time your code to make sure it's faster!). The loop may have too few iterations for vectorization to be profitable; in this case you might want to unroll the loop completely using a directive or pragma to eliminate the loop altogether.

The most common barrier to maximum efficiency of vectorized loops is unaligned data accesses. Loops vectorized using SSE instructions are more efficient when processing vectors aligned to a cache-line boundary. You can force unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the –Mcache_align switch, which is always default with –fastsse. An *unconstrained* data object is a data object that is *not* a common block member and *not* a member of an aggregate data structure. In order for stack-based local variables to be properly aligned, the main program or function must be compiled with –Mcache_align, so it's important to always compile your main program with –fastsse even if it does not contain vectorizable loops. It is important to note that the –Mcache_align switch has no

effect on the alignment of Fortran allocatable or automatic arrays, and that if you have arrays that are constrained, for example vectors that are members of Fortran common blocks, you must specifically pad your data structures to ensure proper cache alignment; –Mcache_align causes only the beginning address of each common block to be cache-aligned. Even when adhering to all of these guidelines, it is of course common for real codes to operate on vectors that begin somewhere in the middle of a data structure, which means the head of the vector may start on an unaligned data address. In these cases, you either must rely on the compiler to maximize alignment through iteration peeling and alternate code generation, or take significant measures to restructure your code in a way that guarantees vectors are aligned in important loops.

By using the information emitted from the –Minfo and –Mneginfo messages, studying loops in your code, and investing a little time to understand the available compiler options that affect vectorization, you can often achieve significant speed-ups in your overall application with a relatively small time investment.

3. Interprocedural Analysis (IPA)

Another important capability of the PGI compilers is interprocedural analysis and optimization (IPA), enabled using **–Mipa**. The –Mipa option must be used at both compile time and link time to enable interprocedural optimization. Optimizations performed by IPA include interprocedural constant propagation, argument removal, pointer disambiguation, alignment detection, alignment propagation, global variable modification and reference detection, F90 shape propagation, function inlining, vestigial function removal, and library optimization. In addition, information gathered through IPA enables more aggressive and precise optimization by other phases of the compiler, including the global optimizer, vectorizer, parallelizer, loop unroller and code generator.

The WUPWISE benchmark provides an example of the type of optimizations performed and enabled by IPA. WUPWISE is a small benchmark, about 2200 lines of FORTRAN 77. It is intended to be representative of computations performed by Quantum ChromoDynamics applications. The following table shows the performance of WUPWISE on an AMD64 processor using 3 different combinations of PGI compiler options:

PGF95 Compiler Options	Execution Time in Seconds
–fastsse	156.49
–fastsse –Mipa=fast	121.65
–fastsse –Mipa=fast,inline	91.72

The large performance improvement obtained by adding **–Mipa=fast** is enabled by interprocedural constant propagation. WUPWISE spends a great deal of time performing matrix multiplication on very small 4x3 matrices of type `COMPLEX*16`. Without IPA enabled, the compiler has no way to determine the size of these matrices, and vectorizes the loops involved in these computations. The overhead of vectorization outweighs the benefits in this case. With IPA enabled, the sizes of these matrices are determined at compile time and propagated throughout the program. As a result, the compiler chooses to completely unroll the matrix multiplication loops, eliminating the loops altogether rather than vectorizing them.

The additional performance improvement obtained with **–Mipa=fast,inline** is achieved by inlining the matrix multiplication subroutine calls and all function calls within them. The result is highly optimized straightline code with no branches or loops required to perform the 4x3 matrix multiplications.

The speed-ups achieved here are obviously very good, and display the potential gains from using IPA. However, in our experience the performance gains for Fortran are more modest, usually averaging 5% - 10% on most codes. For C and C++ applications, automatic pointer disambiguation and the importance of function inlining often lead to more dramatic average speed-ups using IPA.

There are two IPA opimizations that are not performed as part of –Mipa=fast, and which can be very useful. **–Mipa=safe:<name>** declares that the named function or all functions in the named library are *safe*. A safe procedure does not call back into the known procedures and does not change any known global variables. Without –Mipa=safe, any unknown procedures called by or from a given procedure will inhibit IPA optimization of that procedure. In particular, it can be useful to declare certain pre-compiled libraries as safe to ensure they do not inhibit IPA optimization. –Mipa=safeall declares that all unknown procedures are safe; obviously this option must be used with caution.

The second useful option is **–Mipa=libopt**, which allows IPA optimization of routines from pre-compiled static binary libraries. If such a library is created by compiling some or all of the functions in the library with IPA enabled, the IPA information is preserved as part of the object file included in the library. If –Mipa=libopt is specified when linking an application that uses the library, the PGI compilers will include functions from the library in the list of candidates for IPA optimization and will potentially recompile them from an intermediate

representation preserved in the object file. This allows optimizations such as constant propagation into pre-compiled binary libraries, which as we've seen previously can be very effective. If the **–Mipa=libinline** option is specified, the PGI compilers will attempt to inline functions from pre-compiled static binary libraries built using IPA.

4. Function Inlining

The WUPWISE example in section 3, *Interprocedural Analysis*, showed the speed-ups that can occur with IPA-driven function inlining. In addition, the **–Minline** option can be used to specify precisely which routines, or classes of routines, should be inlined (or not). Following is the synopsis of the –Minline option:

```
-Minline[=lib:<name> | <func> | except:<func> |
         name:<func> | size:<n> |levels:<n>]
```

where the sub-options are defined as follows:

```
lib:<inlib>     Inline extracted functions
                from inlib

<func>          Inline function func

except:<func>   Do not inline function func

name:<func>     Inline function func

size:<n>        Inline only functions smaller
                than n statements

levels:<n>      Inline n levels of functions
```

In the description above, `inlib` is an inline library created using the **–Mextract** option, described in detail in the *PGI User's Guide*. Using the other sub-options, it is possible to specify that certain functions should be inlined, exclude certain functions from being inlined, limit inlining to functions smaller than a specified size determined (approximately) by statement count, or limit inlining to a specified number of levels of function call depth. The default, if no sub-options are specified, is to inline all eligible functions up to 3 levels deep.

Note that there are several restrictions on the type of functions that are eligible for inlining, including Fortran main or BLOCK DATA programs, Fortran subprograms with FORMAT statements, or programs with multiple entries. In addition, Fortran functions or subroutines are not inlined into statement functions, or if common block mismatches occur between caller and callee, or if argument mismatches occur, or if a name clash between a function and a local variable occurs.

Restrictions on inlining of C and C++ functions include functions containing switch statements, functions which reference a static variable whose definition is nested within the function, and functions which accept a variable number of arguments. Certain C and C++ functions can only be inlined into the file that contains their definition: static functions, functions which call a static function, and functions which reference a static variable.

In general, it is easiest to use the completely automatic and global cross-file inlining optimizations performed by –Mipa=inline. However, for some applications the –Minline option enables precisely targeted function inlining which can be beneficial. In the absence of using –Mipa=inline, it is recommended that the option –Minline=levels:10 be used by default for best performance of C++ programs.

5. Parallelization for Multi-core Processors

The PGI Fortran, C and C++ compilers all have extensive capabilities in generation of parallel code for multi-core processors. The **–Mconcur** option instructs the compilers to parallelize loops automatically according to default heuristics. The **–mp** option instructs the compilers to interpret directives and pragmas and parallelize applications according to the OpenMP 2.5 parallel programming standard. It is perfectly safe to add OpenMP directives or pragmas and function calls to your application and compile *without* –mp. The OpenMP directives and pragmas will be ignored and the OpenMP function calls resolved for serial execution. This allows incremental parallelization of an application using OpenMP without perturbing ongoing development or use of the serial version. In addition, it is possible to use –Mconcur and –mp together, leveraging both automatic and user-directed parallelization.

Both **–mp** and **–Mconcur** must be used at both compile time and link time to successfully build a parallel executable. As with the optimizations discussed previously, the –Minfo and –Mneginfo options can be used to see which loops are parallelized (and how), and which loops failed to parallelize (and why).

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

available to let you override most of these restrictions on auto-parallelization. The **–Mconcur=cncall** option, and the associated directive/pragma, enable parallelization of loops with procedure calls. The **–Mconcur=innermost** option forces the compiler to parallelize innermost loops.

Auto-parallelization can provide significant performance improvements on applications that are dominated by computationally intensive loops with a lot of data re-use. Consider the following loop from the MGRID benchmark:

```
    DO 10 I3=2,N-1
    DO 10 I2=2,N-1
    DO 10 I1=2,N-1
10  R(I1,I2,I3)=V(I1,I2,I3)
   & -A(0)*(U(I1,I2,I3))
   & -A(1)*(U(I1-1,I2,I3)+U(I1+1,I2,I3)
   &        +U(I1,I2-1,I3)+U(I1,I2+1,I3)
   &        +U(I1,I2,I3-1)+U(I1,I2,I3+1))
   & -A(2)*(U(I1-1,I2-1,I3)+U(I1+1,I2-1,I3)
   &        +U(I1-1,I2+1,I3)+U(I1+1,I2+1,I3)
   &        +U(I1,I2-1,I3-1)+U(I1,I2+1,I3-1)
   &        +U(I1,I2-1,I3+1)+U(I1,I2+1,I3+1)
   &        +U(I1-1,I2,I3-1)+U(I1-1,I2,I3+1)
   &        +U(I1+1,I2,I3-1)+U(I1+1,I2,I3+1) )
   & -A(3)*(U(I1-1,I2-1,I3-1)+U(I1+1,I2-1,I3-1)
   &        +U(I1-1,I2+1,I3-1)+U(I1+1,I2+1,I3-1)
   &        +U(I1-1,I2-1,I3+1)+U(I1+1,I2-1,I3+1)
   &        +U(I1-1,I2+1,I3+1)+U(I1+1,I2+1,I3+1))
```

This is a typical grid routine that performs a stencil operation as it loops over a 3D space. Compiling the function containing this loop nest using –Mconcur results in the following:

```
 resid:
  ...
  189, Parallel code for non-innermost loop
          activated if loop count >= 33; block
          distribution
  191, 4 loop-carried redundant expressions
          removed with 12 operations and 16
          arrays
       Generated vector sse code for inner loop
       Generated 8 prefetch instructions for
          this loop
       Generated vector sse code for inner loop
       Generated 8 prefetch instructions for
          this loop
   ...
```

There is a lot going on here, just for the outer loop. First, note that the compiler parallelizes the outer loop rather than one of the inner loops. The default heuristics are designed to parallelize outermost loops and vectorize innermost loops. Also, the compiler has generated alternate versions of the outermost loop at line 189; a serial version is executed if a dynamic check shows the iteration count is less than 33, and the parallel version is executed if the iteration count is greater than or equal to 33. This threshold for parallel execution (and a little bit of runtime overhead) can be eliminated by compiling with **–Mconcur=noaltcode**. The threshold can be set explicitly on a loop-by-loop basis using directives or pragmas. Finally, the outer loop iterations are allocated in block fashion (rather than round-robin fashion) to the available cores. By default, execution uses only one core; to enable parallel execution, the NCPUS environment variable must be set to the desired number of cores. In future releases of the PGI compilers, it is likely that the number of cores will be dynamically determined at execution time, and multiple cores will be used by default.

In the innermost loop, the compiler has performed loop-carried redundancy elimination to reduce both the number of operations and the number of memory references. Both versions of the innermost loop (the version in the serial outermost loop, and the version in the parallelized outermost loop) are vectorized using SSE instructions, and explicit prefetch instructions for 8 different array references are added for both versions.

With all of these transformations in place, the MGRID benchmark shows an overall speed-up on a dual-core AMD64 processor of about 40% (wall-clock time) over the time required on a single core of the same processor running a true serial version of the benchmark. Unfortunately, MGRID is the exception rather than the rule when it comes to current capabilities in auto-parallelization for multi-core. It is more common with current-generation PGI compilers to see 5% - 15% speed-ups on parallelizable applications using –Mconcur.

In general, loops that are not memory bound, loops or routines that benefit from larger on-chip cache, and loops that are very compute intensive (call transcendental functions, are dominated by floating-point operations, etc) are good candidates for automatic parallelization. You can sometimes improve the effectiveness of auto-parallelization significantly by using it on a carefully targeted set of loops or functions, rather than enabling it globally. PGI is exploring optimizations to reduce parallel startup overhead and broaden the class of loops and code regions that can be parallelized automatically on current and future generation multi-core processors.

OpenMP is widely used on SMP systems like the Cray XD1 to enable parallel execution. As opposed to automatic parallelization, which is intended to provide a quick boost in performance on multi-core processors on certain classes of applications, OpenMP enables very sophisticated parallelization from the loop level up to and including parallelization of an entire application. The PGI Fortran, C and C++ compilers fully support OpenMP 2.5, as documented in detail in the PGI User's Guide. Rather than attempt to explain all of these capabilities here, we leave it as an exercise to the reader to explore both the PGI documentation and the OpenMP 2.5 standard for a more thorough understanding of how this

programming model can be used on Cray XD1 systems and future-generation Cray XT3 systems incorporating multi-core processors.

6. Miscellaneous Optimizations

There are a number of miscellaneous PGI compiler options and directives that can be useful in optimizing applications. Some of these have a large impact on some applications, and no (or even negative) impact on others. We describe these only briefly here, with the expectation that a concise list can serve as a launching point into more detailed descriptions provided in the PGI User's Guide.

The **–Mfprelaxed** option instructs the compiler to perform 32-bit floating-point square root, reciprocal square root, and divide operations using reduced-precision approximations plus Newton's iterations. This optimization is performed by default by a number of x86 compilers. PGI has chosen not to enable it by default as the error introduced versus full 32-bit precision computations can be up to 4 units in the least precision (ULPs) on some operations. However, we are aware of one production oil and gas application that achieves an overall speed-up of over 20% with acceptable precision when this option is used. We have observed similar speed-ups on certain industry-standard benchmark applications.

If your application uses LAPACK, use the **–lacml** link-time option to ensure you link the AMD Core Math Library (ACML). The ACML includes hand-tuned versions of most LAPACK kernels, ensuring near-optimal performance on these commonly-used library calls. In addition to LAPACK, the ACML includes a large and growing body of routines targeted at applications that use FFTs and other common compute intensive algorithms.

The PGI compilers automatically insert explicit prefetch instructions in many vectorized loops. The fetch-ahead distance, maximum number of prefetch instructions per loop, and even the type of prefetch instruction generated can all be controlled at the file level using **–Mprefetch**. For example, –Mprefetch=d:8,n:4 instructs the compiler to set the fetch-ahead distance to 8 cache lines and the maximum number of prefetch instructions per loop to 4. It is sometimes useful to experiment with this option, especially with the fetch-ahead distance. The default fetch-ahead distance is 4 cache lines, which has proved most generally optimal on a range of benchmarks and applications used internal to PGI to validate each release of the compilers. There are exceptions to every rule, and in some cases the overhead of prefetching instructions outweighs the benefit of prefetching. It is sometimes profitable to disable prefetching entirely using **–Mnoprefetch**. Finally, the directive

```
cmem$ prefetch <var1>[,<var2>[,…]]
```

can be used to explicitly fetch individual array elements on a fine-grained loop-by-loop basis. Note that the sentinel for this directive is `cmem$`, which is distinct from the `cpgi$` sentinel used for PGI optimization directives. A similar pragma is available for C and C++. In particular, we have seen cases where prefetching of index vector elements using this directive, which is never performed by default, can provide up to 10% speed-ups on applications that make heavy use of indirection.

The current implementation of exceptions handling in the PGC++ compiler uses setjmp/longjmp in a way that is portable across all operating systems and target processors. PGI is working toward a Linux and AMD64 implementation that is both highly optimized and compliant with the vendor-neutral C++ ABI. In the meantime, applications that do not require exceptions support often speed up dramatically if the **—no_exceptions** option is used. In addition, as noted earlier, it is a good idea to enable either –Mipa=inline or –Minline=levels:10 when compiling C++ applications.

The PGI compilers produce 64-bit executables by default on AMD64 processor-based systems running a 64-bit operating system. In particular, this means that C and C++ pointers and `long` data occupy 64-bits. For some C and C++ applications, this can lead to a significant increase in the aggregate size of the space occupied by such data elements. In some cases, C and C++ applications that do not require 64-bit addressing speed up by 30% – 40% if they are compiled with **–tp k8-32**. This instructs the compiler to generate a 32-bit executable, limiting memory range to a maximum of 2 GBytes. In particular, if you are developing C or C++ applications for the Cray XT3 that use a small amount of local memory on each node, try building with –tp k8-32 to see if it improves performance.

The aggregate option –fastsse includes a number of options that PGI has determined provide generally optimal scalar and SSE vector performance. The options included in –fastsse may change from release to release based on evolving knowledge of the target processors and evolving optimization capabilities of the compilers. The intent is to simplify the process of determining optimal compile flags for any given release or target system. On all platforms and for all releases so far, the –O2 option is the highest level of scalar optimization enabled by default as part of –fastsse. The PGI compilers support a –O3 option, which enables more aggressive hoisting and scalar replacement which are not always profitable. It is sometimes useful to use **–O3** in addition to –fastsse to enable these optimizations, but you should always time your code with and without –O3 to ensure it's beneficial.

Non-temporal store instructions on AMD64 processors bypass the data cache, and can result in improved performance on memory-intensive loops that operate over a very large index range. For example, the STREAM benchmark requires use of non-temporal stores to achieve maximum bandwidth on AMD64 processor-based systems. The PGI compilers perform static compile-time checks of index ranges where possible, and dynamic checks at execution time. If the compiler determines according to heuristics that a loop is likely to benefit from non-temporal stores, it will generate an alternate loop that is executed in such cases.

The **–Mmovnt** option instructs the compiler to generate non-temporal stores and prefetch instructions, even when it cannot determine that this is beneficial. This option should be used carefully, as it can lead to significant slowdowns for some loops. Of more interest and use is the directive

```
cpgi$altcode [ (n) ] nontemporal
```

which allows precise control over the loop iteration count above which non-temporal stores will be used. By default, the compilers will use non-temporal stores if the loop is memory-intensive and the data referenced in the loop does not fit in the level-2 data cache.

Finally, the PGI directory structure is designed so that multiple releases (e.g. 5.2, 6.0, 6.1) can be installed concurrently on the same computer system. Your shell path setting determines which release of the compilers is invoked by default. The **–V[version]** option can be used to specify which version of the PGI compilers should be used to compile a given file, overriding the default. If you are in an environment where PGI 6.1 is default, you can specify that one or more files be compiled with PGI 6.0 simply by adding –V6.0 to the command-line for those files. While many Cray sites provide the modules command to switch back and forth between releases, the –V option has the advantage that it can be used on a file-by-file basis. It allows users to migrate from release-to-release on their own schedule, and to easily test and measure the capabilities of the PGI compiler releases head-to-head on a file-by-file basis. Note that in some cases there are incompatibilities that prevent mixing object files compiled with two different releases in the same executable.

Conclusion

PGI and Cray are working to deliver the best possible compile-and-go performance to Cray XT3 and Cray XD1 users. In many cases, you can obtain near-optimal compiled performance using **–fastsse –Mipa=fast**, or, if inlining is required, by using **–fastsse –Mipa=fast,inline**. However, there are always cases when the programmer knows more about the application and input data than the compiler can ever determine statically. Even profile-feedback optimization, which we didn't address in this paper, is limited in the scope of what it can determine dynamically about an application's execution patterns or data requirements. In such cases, the programmer can improve performance, sometimes dramatically, by experimenting with various options and directives to provide the compiler with information that drives more efficient code generation. We have outlined a number of such options and directives which have proven useful based on PGI's experience with various applications and benchmarks.

About the Authors

Douglas Miles is responsible for all business and technical operations of The Portland Group (PGI). He has worked in various positions over the last 20 years in HPC applications engineering, math library development and technical marketing at Floating Point Systems, Cray Research Superservers, PGI and STMicroelectronics. He can be reached by e-mail at *douglas.miles@st.com*. Brent Leback is the Applications Engineering manager for PGI. He has worked in various positions over the last 20 years in HPC customer support, math library development, applications engineering and consulting at QTC, Axian, PGI and STMicroelectronics. He can be reached by e-mail at *brent.leback@st.com*. David Norton works in strategic technical marketing as a customer liaison for PGI. He has worked in various positions over the last 20 years in application development and technical marketing at the Naval Research Laboratory, Digital, Compaq, Quadrics, LinuxNetworx, and PGI. He can be reached by e-mail at *norton@hpfa.com*.