Benchmark Performance of Different Compilers on a Cray XE6

Michael Stewart and Yun (Helen) He
National Energy Research Scientific Computing Center (NERSC)

ABSTRACT: *There are four different supported compilers on NERSC's recently acquired XE6, Hopper. Our users often request guidance from us in determining which compiler is best for a particular application. In this paper, we will describe the comparative performance of different compilers on several MPI benchmarks with different characteristics. For each compiler and benchmark, we will establish the best set of optimization arguments to the compiler.*

KEYWORDS: XE6, Compilers, Performance.

1. Introduction

1.1 Hopper, the XE6 System Used

The computer platform used in this study is NERSC's new flagship system: a Cray XE6 system, Hopper. Hopper was installed at NERSC in the last quarter of 2010 with early users on the system running codes free of computing hours charge until it was put into production. Hopper entered formal production on May 1, 2011.

Hopper has 6,384 compute nodes, each consists of 2 twelve-core AMD MagnyCours, 2.1 GHz processors, with a total of 153,126 compute cores available for scientific applications serving over 3,000 users from DOE and universities research communities.

Each compute core has a peak performance of 8.4 Gflops resulting in a system with a peak performance of 1.28 Pflops. Most of the compute nodes have 32 GB (384 nodes have 64 GB) DDR3 1333 MHz memory resulting the overall memory available on the compute nodes of about 218 TB. Each core has own 64 KB of L1 cache, and 512 KB of L2 cache, and every 6 cores (one NUMA node) share 6 MB of L3 cache. There are 2 DDR3 1333 MHz memory channels per die (6 cores, called a NUMA node) on the tweleve-core MagnyCours processor. The memory access within a NUMA node (local memory) differs from memory access to a remote NUMA node. Compute nodes are connected via the high-bandwidth, low-latency "Gemini" network in a 3D torus.

1.2 Available Compilers on Hopper

There are four compilers available on Hopper. The user environment for each compiler is provided by Cray via loading a corresponding Programming Environment module. The default compiler is PGI. The other three compilers are Pathscale, Cray, and GNU, respectively.

To use Pathscale Compilers:
```
% module swap PrgEnv-pgi PrgEnv-pathscale
```

To use Cray Compilers:
```
% module swap PrgEnv-pgi PrgEnv-cray
```

To use GNU Compilers:
```
% module swap PrgEnv-pgi PrgEnv-gnu
```

1.3 How Codes are Compiled on Hopper

Cross compilation from login nodes is used to build executables to run on the Hopper compute nodes. To use a particular compiler, first swap to the corresponding PrgEnv module as shown in the previous section. Then use compiler wrappers: "ftn" for Fortran codes, "cc" for C codes, and "CC" for C++ codes, to compile the codes. The wrappers can find the proper system and MPI libraries automatically.

1.4 Compiler Flags Comparison

Table 1 shows the common compiler flags including optimization, IO, OpenMP, fixed or free source format, etc. for different compilers.

Table 1. Common Compiler Options Comparison

PGI	Pathscale	Cray	GNU	Explanation
-fast	-Ofast	-O3	-O3	High level optimization
-mp=nonuma	-mp	-h omp (default)	-fopenmp	Enable OpenMP
-byteswapio	-byteswapio	-h byteswapio	-fconvert=swap	Read files in big-endian
-Mfixed	-fixedform	-f fixed	-ffixed-form	Fixed form source
-Mfree	-freeform	-f free	-ffree-form	Free form source
-V	-dumpversion	-V	--version	Show version info
Not implemented	-zerouv	-e 0	-finit-local-zero	Zero fill uninitialized values

1.5 Motivations

There are four different supported compilers on NERSC's recently acquired XE6, and our users often request guidance from us in determining which compiler is best for a particular application.

In this paper, we will describe the comparative performance of different compilers on several MPI and benchmarks with different characteristics. For each compiler and benchmark, we will establish the best set of optimization arguments to the compiler.

2. Benchmarks Tested

2.1 NERSC6 Application Benchmarks

Table 2 lists 5 of the 7 application benchmarks from the NERSC6 benchmark suite, a representation of typical NERSC workload used for Hopper procurement. Performance results from these 5 applications using different compilers will be presented.

Table 2. Selected NERSC6 Application Benchmarks

Benchmark	Science	Algorithm	Concurrency	Language
GTC	Fusion	PIC, finite difference	2048	F90
IMPACT-T	Accelerator Physics	PIC, FFT	1024	F90
MAESTRO	Astrophysics	Block structured-grid multiphysics	2048	F90
MILC	Lattice Gauge Physics (QCD)	Conjugate Gradient, sparse matrix, FFT	1024	C, Assembly
PARATEC	Material	DFT, FFT,BLAS	1024	F90

2.2 NPB3.3 MPI Benchmarks

Table 3 shows the NPB 3.3 Kernel MPI Benchmarks used in this study, all at a concurrency of 256 processes.

The NAS Parallel Benchmarks (NPB) are a small set of kernel programs and pseudo program designed to help evaluate the performance of parallel supercomputers. The benchmarks are developed and maintained by NASA Advances Supercomputing Division.

Table 3. NPB 3.3 MPI Benchmarks

Benchmark	Full Name	Level
BT	Block Tridiagonal	D
CG	Conjugate Gradient	E
EP	Embarassingly Parallel	E
FT	Fast Fourier Transform	D
LU	Lower-Upper Symmetric Gauss-Seidel	E
MG	MultiGrid	E
SP	Scalar Pentadiagonal	D

3. Recommended Compiler Options

3.1 PGI Compiler

In Chapter 3, Optimizing and Parallelizing, of the PGI Compiler User's Guide [3], the section "Getting Started with Optimizations" recommends "-fast -Mipa=fast" as "a good set of options to use with any of the PGI compilers". In addition, Cray recommends "-Mfprelaxed" which provides additional optimizations at the possible cost of a loss of floating point precision [4].

Our benchmarks runs will compare the performance of runs compiled with these options:

- -fast - A level of optimization which chooses generally optimal flags for the target platform.
- -fast -Mipa=fast - Enables interprocedural analysis and chooses generally optimal interprocedural options for the target platform.
- -fast -Mfprelaxed - Generates relaxed precision code for those floating point operations that generate a significant performance improvement, depending on the target processor.
- -fast -Mipa=fast –Mfprelaxed - The combination of all above options.

3.2 Pathscale Compiler

In Section 6, Basic Optimization, of the Pathscale Compiler Suite User Guide [5], 6.5 "Compiler Flag Recommendations", Pathscale recommends this optimization strategy, "start tuning with -O2, then -O3, then -O3 -OPT:Ofast and then -Ofast." Cray recommends –Ofast [4].

Our benchmarks runs will compare the performance of runs compiled with these options:

- -O2 - This turns on extensive, but conservative optimizations which are virtually always beneficial, avoiding changes which affect such things as floating point accuracy.
- -O3 - This turns on aggressive optimizations which are generally beneficial but may hurt performance and require extensive compile times.
- -O3 -OPT:Ofast - This adds optimizations which may affect floating point accuracy due to rearrangement of computations.
- -Ofast - This adds interprocedural analysis to the optimizations above.

3.3 Cray Compiler

Cray recommends using the default optimization ("-O2"), which is equivalent to the higher levels of optimization with other compilers. In addition, the "-O3" and "-Ofp3" options can improve performance on some codes [4].

Our benchmarks runs will compare the performance of runs compiled with these options:

- default (-O2)
- -O3
- -Ofp3 - This gives the compiler maximum freedom to optimize floating-point operations, even at the expense of not conforming to the IEEE floating point standard.
- -O3,fp3 – The combination of the above two options.

3.4 GNU Compiler

NERSC has found that for this compiler, “-O3” produces well optimized code for many benchmarks. In addition, Cray recommends these options for additional performance optimizations: “-ffast-math” and “-funroll-loops” [4].

Our benchmarks runs will compare the performance of runs compiled with these options:

- -O3 - This compiles with a high level of optimization.

- -O3 -ffast-math - This performs optimizations at the expense of an exact implementation of IEEE or ISO rules/specifications for math functions.

- -O3 -funroll-loops - This unrolls loops whose number of iterations can be determined at compile time or upon entry to the loop. It also turns on complete loop peeling (i.e. complete removal of loops with a small constant number of iterations). This option makes code larger, and may or may not make it run faster.

- -O3 -ffast-math -funroll-loops - The combination of all above options.

4. Performance Results

4.1 PGI Compiler Options

Figure 1 shows the benchmark performance results using different PGI compiler optimization options discussed in Section 3.1.

The wall clock time of each run will be normalized against the "-fast" time, so if a job compiled with "-fast" completes in 20 seconds, a job that completes in 23 seconds would be shown on the graph as 1.15, and one that completes in 18 seconds would be shown as .9.

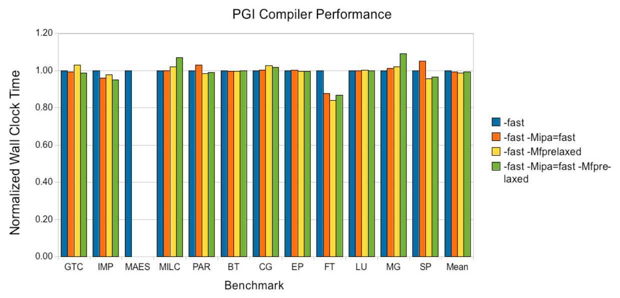

Figure 1. Benchmark performance results using different PGI compiler options.

Note: The Maestro benchmark does not run successfully with optimization options beyond "-fast".

Generally speaking, the other options do not significantly improve the performance over that obtained with "-fast", and in some cases worsen it. The only significant exception to this is the NPB FT Level D benchmark whose performance is greatly improved by each of the other three options.

4.2 Pathscale Compiler Options

Figure 2 shows the benchmark performance results using different Pathscale compiler optimization options discussed in Section 3.2. The wall clock time is normalized against the "-O2" time.

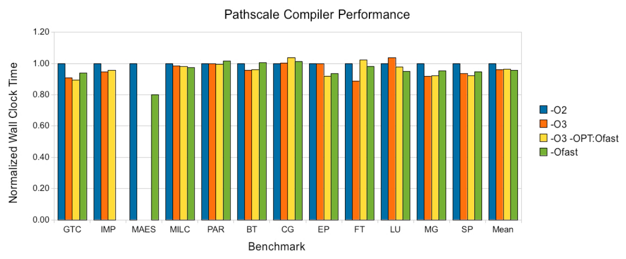

Figure 2. Benchmark performance results using different Pathscale compiler options.

Note: The Impact benchmark does not run with “-Ofast”. The Maestro benchmark gets such poor performance with the “-O3” and "-O3 -OPT:fast" optimizations that the graph and mean would be seriously distorted.

The Pathscale compiler does not optimize very well with “-O2” compared to the more aggressive optimizations. “-O3” optimizes almost every code quite well. In general, the extra options do not improve performance significantly.

4.3 Cray Compiler Options

Figure 3 shows the benchmark performance results using different Cray compiler optimization options discussed in Section 3.3. The wall clock time is normalized against the default ("-O2") time.

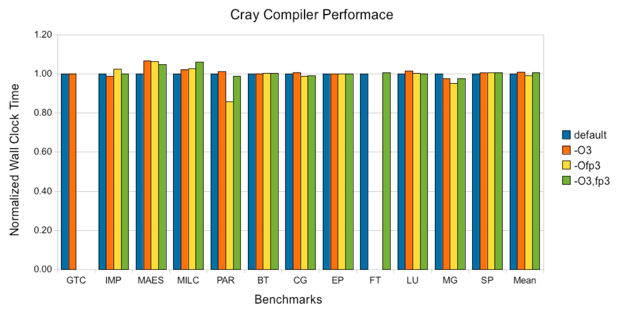

Figure 3. Benchmark performance results using different Cray compiler options.

Note: The GTC benchmark does not run when compiled with "-Ofp3" as one of the optimizations. The FT Level D benchmark has much worse performance with "-O3" and "-O3,fp3" options than with the other two, so much so that including them in the graph would seriously distort the mean.

Only one of the benchmarks shows a significant improvement over the default optimization, Paratec with "-Ofp3". For all the other benchmarks, the higher levels of optimization give little or no improvement in performance.

4.4 GNU Compiler Options

Figure 4 shows the benchmark performance results using different GNU compiler optimization options discussed in Section 3.4. The wall clock time is normalized against the "-O3" time.

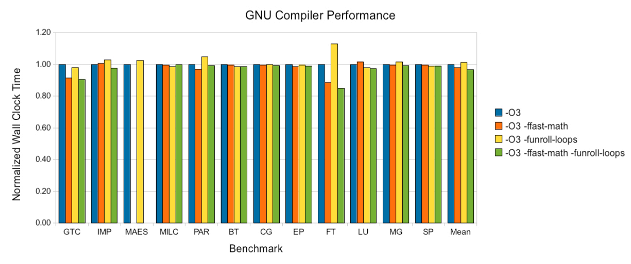

Figure 4. Benchmark performance results using different GNU compiler options.

Note: The Maestro benchmark does not run successfully when compiled with the "-ffast-math" option.

"-O3" generally give a good level of optimization, but it seems to be worthwhile to try the "-ffast-math" option, since in many cases it does improve a code's performance significantly.

4.5 Overall Compiler Comparisons

In this section, for each benchmark, the best results for each compiler regardless of the optimizations that produced them, are compared against each other. Figure 5 shows the performance results for all the benchmarks from each compiler. The results are normalized against the PGI compiler.

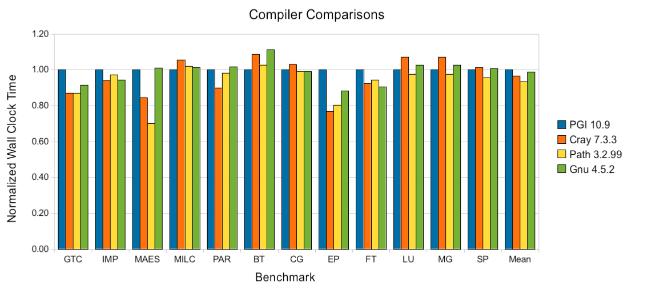

Figure 5. Benchmark performance results using different compilers.

The Pathscale compiler gives the fastest performance for 6 out of the 12 benchmarks; The Cray compiler's gives the fastest performance on 3 of the 12. The GNU compiler is fastest 2 of the 12, and the PGI compiler is fastest 1 of the 12.

The Cray and Pathscale compiled codes run on the average about 5% faster than the PGI compiled codes, and the GNU compiled codes have on the average about the same performance as the PGI compiled codes, although there are wide variations with individual benchmarks.

The Cray compiler is a relatively new product on AMD based systems, and we have noticed a steady improvement in its performance with newer releases. This compiler is only available on Cray systems, so if you port to another architecture, you would need to choose another compiler and develop another Makefile.

The Pathscale compiler is available on other Intel and AMD platforms, but it is not very common, so it is very likely that any port to another system will require a different compiler and a new Makefile.

PGI is by far the most commonly available commercially produced compiler, and a PGI Hopper Makefile would require few changes to run on the many systems on which PGI is available. The recommended "-fast" option can have different optimizations depending on the underlying architecture on which it is compiled, and it should produce well optimized code on any platform.

The GNU compiler is available for almost any Linux or Unix platform, and a GNU Hopper Makefile would also require few changes on other platforms, but there is no guarantee that its relatively good performance on Hopper would be repeated on other platforms.

5. Summary

In order to achieve best performance results, users should experiment with different compilers and compiler options to tune their applications on Hopper.

On the average the Pathscale and Cray compilers produce somewhat faster code on Hopper (and other Cray systems), since they are specifically designed for these processors. In addition the Cray compilers make use of the Cray math libraries at compile time to further optimize codes.

PGI compilers are available on a wide variety of platforms besides Crays and produce well optimized codes on all platforms. A PGI targeted Makefiles on Hopper would work with few changes on many different platforms.

Using the gnu compilers allows you to compile on virtually every Unix and Linux system. Although the performance on Hopper for some codes with GNU compilers is quite good, there is no guarantee for optimal performance on other platforms.

Acknowledgments

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

References

1. NERSC Web Page on Hopper: http://www.nersc.gov/users/computational-systems/hopper/

2. NPB Parallel Benchmarks. http: //www.nas.nasa.gov/Resources/Software/npb.html

3. PGI Compiler User’s Guide. Release 2011. http://www.pgroup.com/doc/pgiug.pdf

4. Jeff Larkin. “XE6 Porting and Tuning Tips”. Using the Cray XE6 Workshop, Feb 7-8, 2011. NERSC. Oakland, CA.

5. Pathscale Compiler Suite User Guide. Version 3.2. http://www.pathscale.com/docs/UserGuide.pdf

About the Authors

Both Michael Stewart and Helen He are High Performance Computing Consultants at NERSC. Email: pmstewart@lbl.gov, yhe@lbl.gov.