The Effects of Compiler Optimizations on Materials Science and Chemistry Applications at NERSC

Megan Bowling, Zhengi Zhao, and Jack Deslippe
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA

E-mail: {mbowling, zzhao, jrdeslippe}@lbl.gov

Abstract: **Materials science and chemistry applications consume around 1/3 of the computing cycles each allocation year at NERSC. To improve the scientific productivity of users, NERSC provides a large number of pre-compiled applications on the Cray XE6 machine, Hopper. Depending on the compiler, compiler flags and libraries used to build the codes, applications can have large differences in performance. In this paper, we compare the performance differences arising from the use of different compilers, compiler optimization flags and libraries available on Hopper over a set of materials science and chemistry applications that are widely used at NERSC. The selected applications are written in Fortran, C, C++, or a combination of these languages, and use MPI or other massage passing libraries as well as linear algebra, FFT, and global array libraries. The compilers explored are the PGI, GNU, Cray, and Intel compilers. It should be emphasized that the compiler optimizations ultimately adopted for production builds are optimizations where the resulting binaries pass strict validity checks and can be used for scientific calculations. These builds do not necessarily represent the highest optimizations the compilers can reach.**

Keywords-component; compilers; optimization; libraries; performance; applications; FFT; BLAS; LAPACK; LibSci; ACML

I. INTRODUCTION

Materials science and chemistry applications make up approximately 1/3 of the NERSC workload [1]. In order to increase the productivity of our users, NERSC supports a set of pre-compiled materials science and chemistry programs that scientists typically use when conducting research in these fields. This effort is designed to save time for the researchers as well as to save compute cycles by ensuring that researchers are using an optimized version of each code.

Fully optimizing a given application's performance often requires creating a full profile for a typical run and modifying the source based on the results. However, in many cases, significant performance gains can be achieved by simply optimizing the code over the matrix of possible compilers, compiler options and libraries available. In this paper, we explore the performance variability of six common materials science applications at NERSC with respect to this matrix of internal and external compile time options.

NERSC has a set of four different compilers, PGI, Intel, GNU and Cray, that we will explore on Hopper [2], NERSC's Cray XE6 system, each of which could potentially produce different performance results [3]. Additionally, materials science applications generally rely heavily on math libraries such as FFTW, BLAS, LAPACK and ScaLAPACK. NERSC provides several library options for these routines: FFTW2, FFTW3, LibSci, ACML and MKL. In this paper, we compare the performance of VASP [4], Quantum ESPRESSO [5], NAMD [6], LAMMPS [7], BerkeleyGW [8] and NWChem [9] with the compilers and libraries listed above.

II. MATERIAL AND METHODS

A. Hopper, Cray XE6

All compilations were run on Hopper, NERSC's Cray XE6 peta-flop system. Files were stored on the local Lustre scratch file system. Tests were mainly conducted on Grace, Hopper's test system, in order to achieve a greater rate of reproducibility in performance numbers due to Grace's reduced level of contention. Grace has 12 nodes, 288 cores only. Those large concurrency tests were done on Hopper. To reduce the runtime noise, each benchmark test was run three times, and the best result was taken for each compiler.

B. Compilers Available on Hopper

There are five different compilers available to build applications on Hopper: PGI, Intel, Pathscale, Cray and GNU. In this study, Pathscale was excluded primarily because future support for Pathscale will be limited on Hopper. The Cray compiler was also excluded from the NAMD, LAMMPS, and NWChem builds because there were irresolvable error messages generated during compilation or run time.

C. Libraries

NERSC has a set of 13 math libraries to provide the math routines that are required by the materials science and chemistry applications in order to perform their greater function. Table 1 lists the math libraries used in the optimization of these applications.

Table 1 NERSC Math Libraries

Library	Description
ACML (AMD core math library)	A full implementation of Level 1, 2 and 3 Basic Linear Algebra Subroutines (BLAS), with key routines optimized for high performance on AMD Opteron™ processors. A full suite of Linear Algebra (LAPACK) routines. As well as taking advantage of the highly-tuned BLAS kernels, a key set of LAPACK routines has been further optimized to achieve considerably higher performance than standard LAPACK implementations. A comprehensive suite of Fast Fourier Transforms (FFTs) in both single-, double-, single-complex and double-complex data types. Random Number Generators in both single- and double-precision.
FFTW (Fastest Fourier transform in the west)	FFTW is a C subroutine library for computing the discrete Fourier transform (DTF) in one ore more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms).
MKL (Intel math kernel library)	MKL contains highly optimized, extensively threaded math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms, Vector Math, and more.
LibSci	The Cray scientific libraries package is a collection of numerical routines optimized for best performance on Cray Systems. The package includes: Blas, BLACS, LAPACK, ScaLAPACK, IRT, CRAFFT, FFT, FFTW2, and FFTW3.

D. Procedure

The builds and tests for each application followed the same general structure. The compiler optimization flags for each application were chosen mainly based on the developer suggested optimization flags, and also referred to the NERSC recommended compiler optimization flags [10]. For the large concurrency tests with VASP, we ran the jobs on Hopper, the production system. To reduce the noise in the runtime, we ran each test three times, and used the best performance result among the three runs.

Each run from each application was validated against a well-trusted result. In the case of the density functional theory (DFT) codes, we compare the values of the total energies and individual electron eigenvalues. In the case of BerkeleyGW, we validated the results using key terms in the dielectric matrix. Optimized builds that fail to pass this validation are explicitly marked in the below sections. Additionally, we test each application build at a range of MPI tasks to confirm the performance observed from one set of tests is sustained over the different concurrencies.

III. Pre-Compiled Applications Tested

A. VASP (Vienna Ab initio Software Package) 5.2.12

VASP [4] is a DFT program that computes approximate solutions to the coupled electron Kohn-Sham equations for many-body systems. The code is written in Fortran 90 and MPI. Plane waves basis sets are used to express electron characteristics such as electron wavefunctions, charge densities, and local potentials. Pseudopotentials are used to describe the interactions between electrons and ions. The electronic groundstate electronic configuration is determined using one of two diagonalization algorithms: RMM-DIIS (inversion) and Davidson (blocked).

The VASP benchmark was performed using three test cases provided by NERSC users. The first test system contains 155 atoms, and we tested the two commonly used iteration schemes, the RMM-DIIS and the blocked Davidson schemes. We also tested two BLAS/Lapack libraries, LibSci and ACML to identify the VASP performance gain from varying the BLAS/Lapack libraries. The default BLAS/Lapack library for VASP is LibSci. We also tested VASP with another two benchmark cases, which are provided by NERSC users as well, with the mixed iteration schemes for a system with 660 atoms, and with a hybrid calculation for a system containing 105 atoms. This is to check if the performance sustains itself over the different job types, concurrencies and system sizes. Table 2 shows a list of the compiler optimization flags used for each compiler.

Table 2 VASP Compiler Flags

Compiler	Optimization Flags
PGI	O3, -Mvect, -fastsse
Cray	-O ipa0
Intel	-O3, -fast
GNU	-O3, -ffast-math

B. Quantum ESPRESSO (opEn-Source Package for Research in Electronic Structure, Simulation and Optimization) 4.3.2

Quantum Espresso [5] is a Fortran 90 material science program that performs electronic structure calculations and materials modeling at the nanoscale level. These atomistic simulations are found using density-functional theory (DFT), a plane waves (PW) basis set, and pseudopotentials. Tests were run using the Davidson (iterative with overlap matrixes) and conjugate gradient (CG) diagonalization algorithms,

The default FFT library for QE is FFTW3. However, tests were also run using FFFTW2, which is an internal library. We tested ACML and LibSci linear algebra libraries. Table 3 lists the compiler flags that were used for Quantum ESPRESSO.

Table 3 QE Compiler Flags

Compiler	Flags
PGI	-O3, -r8, -mp, -Mpreprocess, -fast, -Mcache_align
Cray	-s real64
Intel	-O3, -r8, -openmp
GNU	-O3, -ffast-math, fdefault-real-8, -fdefault-double-8, -fopenmp

In the QE benchmark, we perform a self-consistent field (SCF) calculation on the (8,0) single walled-carbon nanotube (SWCNT) with an 80 Ry wave-function cutoff in an 11041 au^3 unit cell. It should be noted that this calculation is dominated by the FFT step.

C. NAMD (Molecular Dynamics) 2.8

NAMD [6] is a C++ chemistry application that performs molecular dynamic simulations that compute atomic trajectories by solving equations of motion numerically using empirical force fields. The Particle Mesh Ewald algorithm provides a complete treatment of electrostatic and Van der Waals interactions.

NAMD uses the single precision FFTW2 libraries exclusively (no FFTW3 interfaces). Therefore, we limited our studies to variations in the compiler and the processor core count. The Benchmark used for NAMD was the standard APoa1, 92,424-atom system. Table 4 shows the compiler flags used for NAMD.

Table 4 NAMD Compile Flags

Compiler	Flags
PGI	-fast
Cray	Build Failed
Intel	-ip, -02
GNU	-03, -ffast-math, -fexpensive-optimizations, -fomit-frame-pointer

We used the standard benchmark Apoa1 (92,224 atoms, 12A cutoff + PME every 4 steps) in the test. We measured the time to complete the first 10000 MD steps. We ran the tests over a range of different core counts.

D. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) Mar 15, 2012

LAMMPS [7] is a C++ classical large-scale molecular dynamics code. It computes Newton's equations of motion for systems of particles in a liquid, solid, or gaseous state.

As in the case of NAMD, only the compiler itself was varied in the tests. Each compiler was tested using the four LAMMPS benchmark problems. A description of these algorithms can be seen in Table 5. Table 6 shows the compiler optimization flags used.

Table 5 LAMMPS Algorithms

Algorithm	Description
LJ	Atomic fluid, Lennard-Jones potential with 2.5 sigma cutoff (55 neighbors per atom), NVE integration.
Chain	Bead-spring polymer melt of 100-mer chains, FENE bonds and LJ pairwise interactions with a 2^(1/6) sigma cutoff (5 neighbors per atom), NVE integration.
EAM	Metallic solid, Cu EAM potential with 4.95-Angstrom cutoff (45 neighbors per atom), NVE integration.
Rhodo	Rhodospin protein in solvated lipid bilayer, CHARMM force field with a 10 Angstrom LJ cutoff (440 neighbors per atom), particle-particle particle-mesh (PPPM) for long-range Coulombics, NPT integration.

Table 6 LAMMPS Compiler Flags

Compiler	Flags
PGI	-fastsse,
Cray	Build Failed.
Intel	-O3, -ip, -unroll0
GNU	-O3, -m64, -fexpensive-optimizations, -ffast-math

E. BerkeleyGW 1.0.x

BerkeleyGW [8] is a Fortran 90 material science application for calculating the spectroscopic, or excited state, properties of materials starting from DFT inputs. We limited our tests to the Epsilon executable, which computes the dielectric matrix from a set of DFT orbitals.

We varied the processor cores, compilers and libraries across runs. BerkeleyGW's default FFT library is set to FFTW2, with FFTW3 as an option. We tested both the LibSci and ACML linear algebra libraries. No internal program variables were tested. Table 7 shows a list of the compiler flags used.

In the BerkeleyGW benchmark calculation we consider again the (8,0) SWCNT with a 80 Ry.

wavefunction cutoff, 12 Ry. dielectric cutoff and 240 empty states.

Table 7 BerkeleyGW Compiler Flags

Compiler	Optimization Flags
PGI	-fast
Cray	default
Intel	-fast
GNU	-O3, -ffast-math, -fexpensive-optimizations

F. *NWChem 6.1*

NWChem [9] is a chemistry application that provides tools for scientists and is designed to be scalable and functional on high performance, parallel compute systems. It is a Fortran code, and its parallelization is implemented with Global Array mainly. It can perform quantum mechanic functions, classical functions, hybrid functions, potential energy surface analysis, and electronic structure analysis.

We tested with three test cases chosen from the standard NWChem distribution, dft_semiddirect.nw, tce_active_ccsdt.nw, and tce_polar_ccsd_big.nw. The first performs density functional theory, and the last two perform coupled cluster calculations. We tested the performance difference when using different compilers. The code is linked to the ScaLAPACk and BLAS routines from LibSci. Table 8 shows a list of the compiler optimization flags used.

Table 8 NWChem Compiler Flags

Compiler	Flags
PGI	-Kieee, -fast, -fastsse, -O3, -Mipa=fast
Cray	Build Failed
Intel	-O3, -prefetch, -unroll
GNU	-O3, -mfpmath-sse, -ffast-math

IV. Results

A. *VASP*

We ran VASP that were compiled with different compilers over three selected test cases provided by NERSC users (as described in the previous section). To check the compiler performance consistency over different use scenarios, we ran VSAP over different problem sizes, job types, iteration schemes and the number of processor cores. Figures 1-3 show the test results for the three test cases, respectively. The default build of VASP on Hopper used the PGI compiler and was linked to the LibSci library. Fig. 1a) was computed using the DIIS-RMM algorithm and Fig. 1b) was computed using the Davidson algorithm.

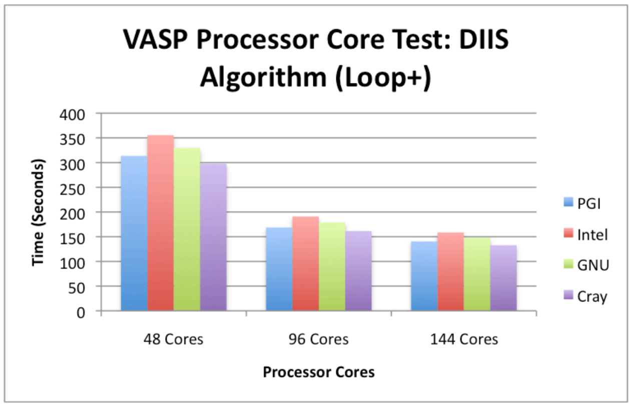

Figure 1. a) VASP compiler performance results found by varying processor cores for the DIIS-RMM algorithm on Grace using a test case containing 155 atoms.

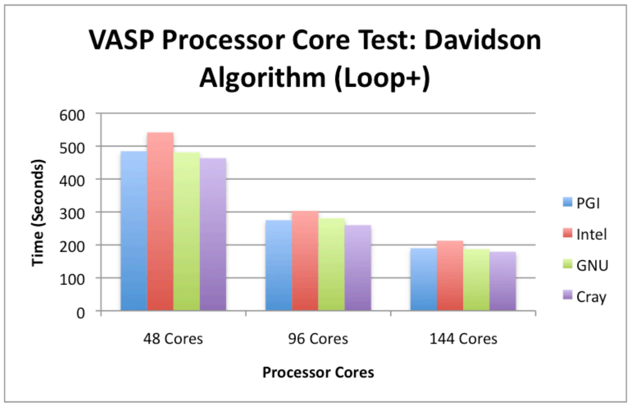

Figure 1. b) VASP compiler performance results found by varying processor cores for the Davidson algorithm on Grace using a test case containing 155 atoms.

Figures 1a) and 1b) show that the Cray compiler consistently outperforms other compilers and is faster than the default compiler, PGI, by up to 5.8% for this small~medium sized VASP runs. Fig. 2 shows that the Cray compiler outperforms other compilers again and has up to 11.2% of speedup compared to the PGI compiler for this larger test case. And, for the third test case, with the hybrid calculation, the Cray compiler and PGI compiler have the same performance. Among all three tests, the Intel compiler was the slowest. It is slower by up to 12% compared to the PGI compiler.

Fig. 4 shows the performance results when VASP was built using the ACML and LibSci libraries. VASP runs slightly faster with ACML library compared to LibSci for VASP. However, the differences across compilers is significantly greater than the differences between libraries.

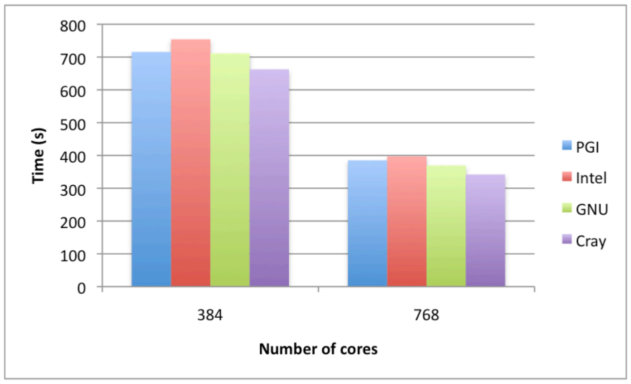

Figure 2. VASP compiler performance results found by varying processor cores for the mixed iteration scheme (DIIS-RMM+Davidson algorithm) on hopper using a larger test case containing 660 atoms. The data shown here were the best run results over 3 repeated runs to reduce the run time noise due to contension from other users on Hopper.

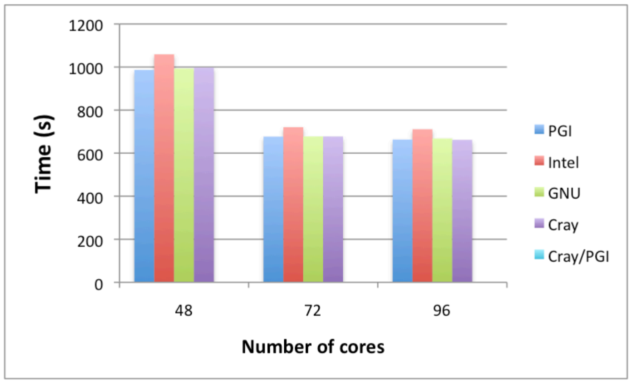

Figure 3. VASP compiler performance results found by varying processor cores for a hybrid calculation on Grace using a test case containing 105 atoms.

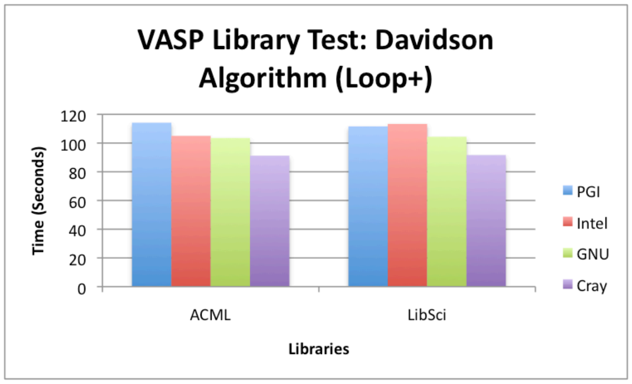

Figure 4. VASP copiler performance results found using different libraries for the Davidson algorithm at 96 cores. The first benchmark case with 155 atoms was used for this test.

It should be noted that it was difficult to compile VASP code with the Cray compiler. We had to modify many palces in the source codes to fix the syntax errors reported by the Cray compiler while other compilers, GNU, Intel and PGI compilers, were all fine with the same syntax used in the code.

B. QE

The results of our test on QE can be seen in Figures 5-7. The default build of QE on Hopper used the PGI compiler, and used the FFTW3 library. In all tests, we allowed for full convergence of the electron charge density. Figures marked a) were run using the Davidson algorithm and Figures marked b) were run using the CG algorithm.

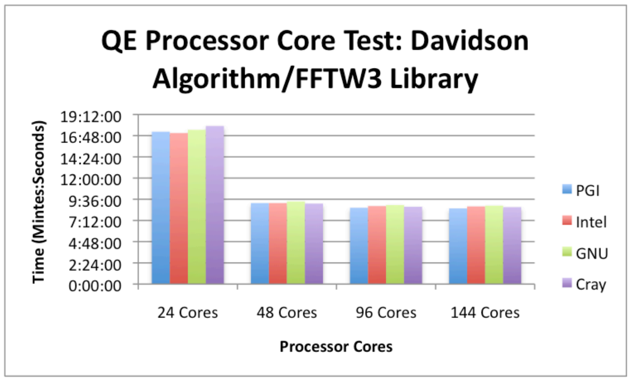

Figure 5. a) QE compiler performance results found by varying the processor cores for the Davidson Algorithm on Grace.

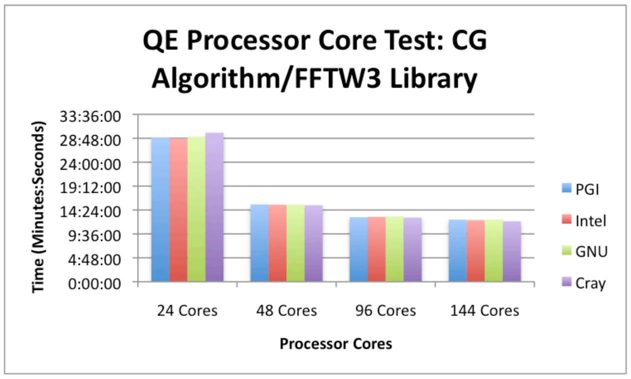

Figure 5. b) QE compiler performance results found by varying the processor cores for the CG algorithm on Grace.

In Figures 5a) and 5b) there is a clear plateau in the performance results when the processor core count is greater than 48. However, the results between compilers were too similar to provide distinct conclusions.

Figures 6 a) and b) show that one obtains significantly better performance with the older internal version of FFTW than with the newer FFTW3 version on Hopper (module fftw/3.2.2.1)

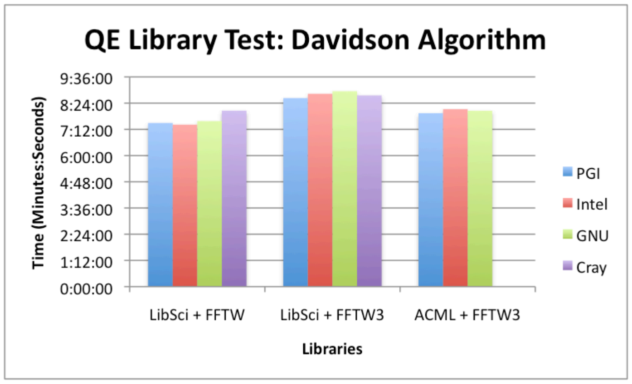

Figure 6. a) QE compiler performance results found using different libraries for the Davidson algorithm at 96 cores..

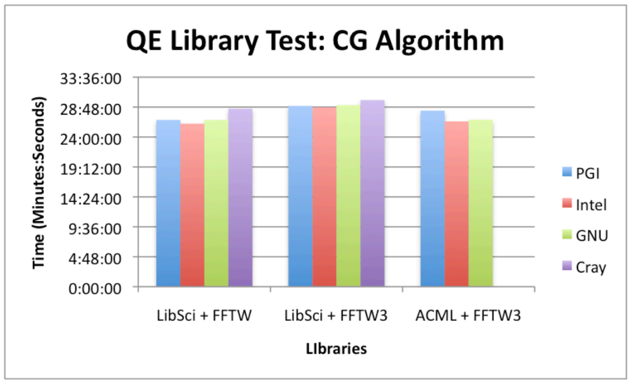

Figure 6. b) QE compiler performance results found using different libraries for the CG algorithm at 24 cores.

C. NAMD

The results of NAMD tests using the standard benchmark Apoa1 (92L atoms, PME) can be seen in Fig. 7. The current build of NAMD on Hopper used the

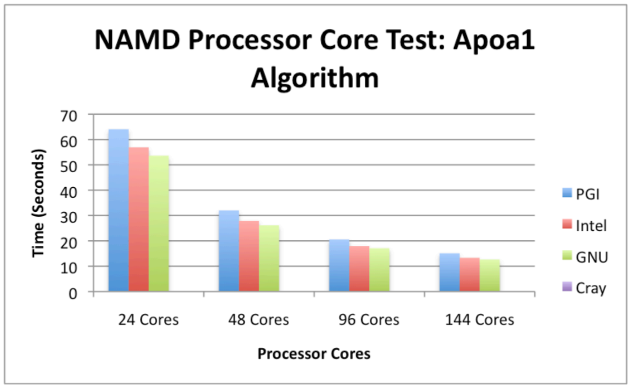

Figure 7. NAMD Compiler performance results found by varying the processor cores for the NAMD standard benchmar Apoa1 on Grace.

GNU compiler and used the single precision FFTW2 from Cray provided library (module fftw/2.1.5.3). We found that the GNU compiler is the optimal compiler for NAMD (C++ code). Note, the code failed to compile with Cray compiler.

D. LAMMPS

The results of LAMMPS test can be seen in Fig. 8. The tests were run multiple times using the four different benchmark test cases provided in the LAMMPS standard distribution.

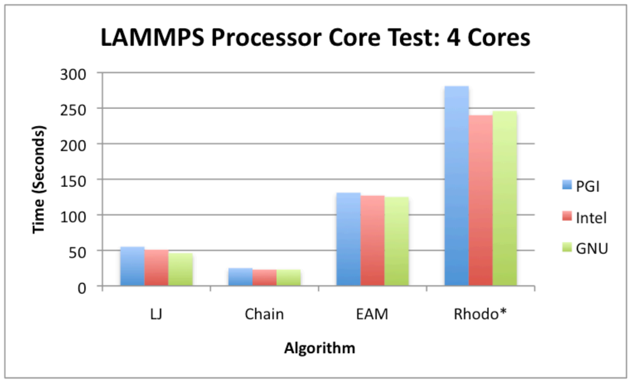

Figure 8. LAMMPS compiler performance results found by testing the four benchmark algorithms at 4 cores. Note that the run time for Rhodo shown in the figure is the actual runtim of Rhodo -600 seconds to fit the Rhodo time into the same figure of other test cases.

It was observed that, for the LJ, Chain, and EAM test cases, the GNU compiler performs slightly better than Intel compiler, but for the test case Rhodo, the intel compiler performs slightly better than GNU compiler. The PGI compiler performs the worst in all test cases.

E. BerkeleyGW

The results of BerkeleyGW tests can be seen in Figures 9 and 10. The default compiler on Hopper was PGI and the default library was FFTW2. The library tests were run using 96 processor cores. From Fig. 8, we can see the expected decrease in computational time as we

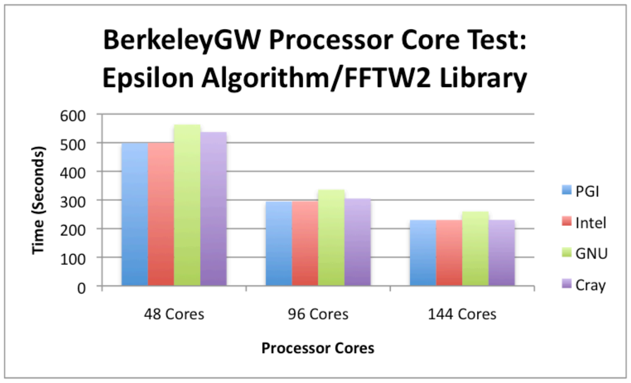

Figure 9. BGW compiler performance results found by varying the processor cores for the Epsilon executable on Grace.

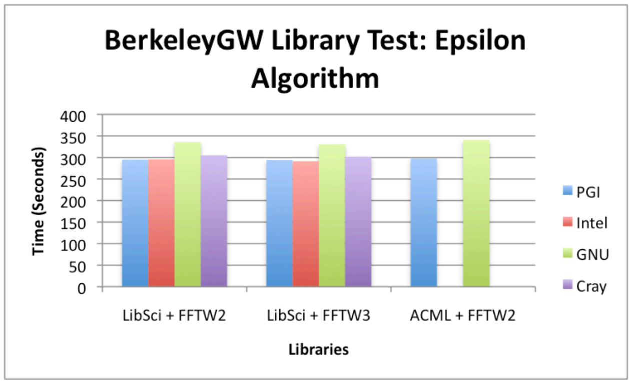

Figure 10. BGW compiler performance results found using different libraries for the Epsilon algorithm at 96 cores.

increase the number of processor cores. The compiler performance does not vary with core count with PGI and Intel consistently giving the best results.

The best overall performance was obtained using the Intel and PGI compilers with the FFTW3 library.

F. *NWChem*

Figures 11-13 show the results of the NWChem tests.

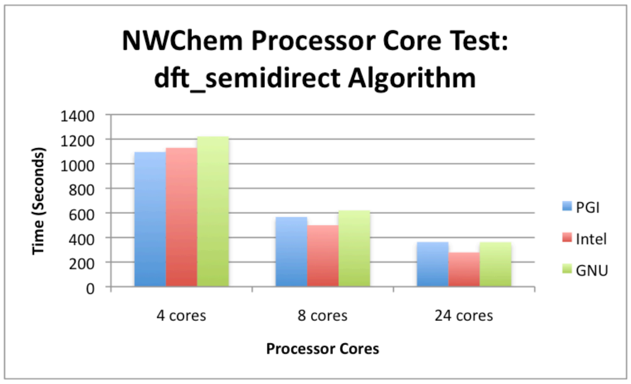

Figure 11. NWChem compiler performance results found by varying the processor cores for the dft_semidirect test case on Grace.

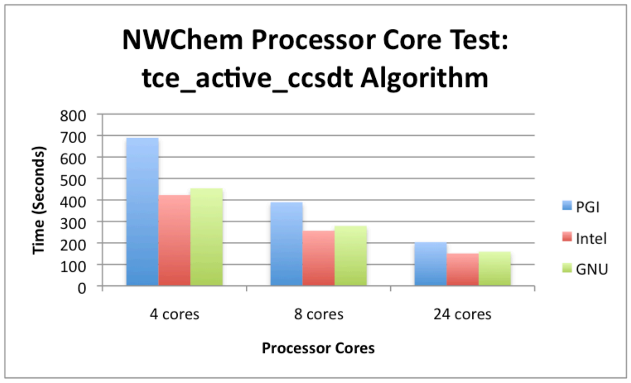

Figure 12. NWChem compiler performance results found by varying the processor cores for the tce_active_ccsdt test case on Grace.

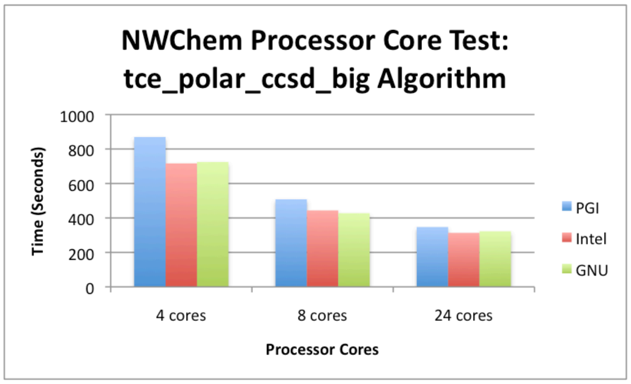

Figure 13. NWChem compiler performance results found by varying the processor cores for the tce_polar_ccsd_big test case on Grace.

The current build of NWChem on Hopper used the PGI compiler. The program was linked to the optimized math libraries BLAS and SCALAPACK from LibSci.. For each test case, the Intel compiler delivers the best performance results. GNU also gave better performance results than PGI, however, it did not perform as well as Intel except on one occasion (see tce_polar_ccsd_big at 8 cores). Overall, Intel outperformed the other two compilers on the majority of the tests run for this application.

Once again, it should be noted, that compiling NWChem code with the Cray compiler was very difficult. After numerous changes in the configure script, makefiles and source codes, we were able to build the NWChem code. Unfortunately, the binary failed to run, throwing an error indicating invalid Fortran binding of C codes. We noticed that the difficulty of using Cray compiler comes from the fact that many existing software packages do not support Cray compilers in their configure scripts and makefiles. In addition, Cray compiler has a more strict syntax check (i.e. follows the standard more rigorously), and hence fails to compile common but non-standard syntax that builds with other compilers. This makes compiling existing software packages significantly more difficult with Cray compilers. However, there is a benefit to rigorous syntax checking in the Cray compiler: we were able to identifiy a bug in the NWChem code.

V. Conclusion

We find that varying the compiler, compiler options and libraries can indeed make significant differences in performance in materials science and chemistry applications on Hopper. Total wall-times typically vary around 10% between optimized builds from each compiler. The best compiler and library option is not universal across the class of applications studied. We summarized the results in Table 13. The last column in the Table shows the performance increase compared to the current build.

Table 13 Summaries of Results

Program	Default		Best Result		Perf. Increase
	Compiler	Library	Compiler	Library	
VASP 5.2.12	PGI 11.9.0	LibSci 11.0.03	Cray 8.0.1	LibSci	11.2%
QE 4.3.2	PGI 11.9.0	FFTW3 3.2.2.1	Intel 12.1.2.273	FFTW 1.2	13.6%
NAMD 2.8	PGI 11.9.0	FFTW 2.1.5.3	GNU 4.6.2	-	0.0%
LAMMPS 03/15/2012	PGI 11.9.0	FFTW 2.1.5.3	Intel 12.1.2.273 GNU 4.6.2	-	0.0%
BGW 1.0.x	PGI 11.9.0	FFTW2 2.1.5.3	Intel 12.1.2.273 PGI 11.9.0	FFTW3 3.2.2.1	1.2%
NWChem 6.1	PGI 11.9.0	libsci 11.0.03	Intel 12.1.2.273	-	34.2%

ACKNOWLEDGMENT

The first author would like to thank Katie Antypas who mentored her during her internship, and David Turner, Mike Stewart and other User Services Group members at NERSC who helped her to get started at NERSC. The work of the first author was supported by the Center for Science and Engineering Education at Lawrence Berkeley National Laboratory and the Student Undergraduate Laboratory Internship of the U.S Department of Energy. The work of the second and third authors were supported by the ASCR Office in the DOE, Office of Science, 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] Francesca Verdier, Code analysis in AY 2011, An internal communication at NERSC.

[2] http://www.nersc.gov/users/computational-systems/hopper/

[3] M. Stewart, Y. He, "Benchmark Performance of Different Compilers on a Cray XE6" in *Cray User Group 2011*, Fairbanks, AK, 2011, pp. 1-6.

[4] http://www.vasp.at/

[5] http://www.quantum-espresso.org

[6] http://www.ks.uiuc.edu/Research/namd/

[7] http://lammps.sandia.gov/

[8] Jack Deslippe, Georgy Samsonidze, David A. Strubbe, Manish Jain, Marvin L. Cohen, and Steven G. Louie, "BerkeleyGW: A Massively Parallel Computer Package for the Calculation of the Quasiparticle and Optical Properties of Materials and Nanostructures," Comput. Phys. Commun. 183, 1269 (2012)

[9] http://www.nwchem-sw.org/

[10] http://www.nersc.gov/users/software/compilers/