Optimizing Utilization across XSEDE Platforms

Haihang You
National Institute for Computational Sciences
Oak Ridge National Laboratory, Oak Ridge, TN 37831
hyou@utk.edu

Charng-Da Lu
Center for Computational Research
University at Buffalo, Buffalo, NY 14203
charngdalu@yahoo.com

Ziliang Zhao
Joint Institute for Computational Sciences
University of Tennessee, Knoxville, TN 37996
zzhao7@utk.edu

Fei Xing
Mathematics Department
University of Tennessee, Knoxville, TN 37996
xing@math.utk.edu

ABSTRACT

HPC resources provided by XSEDE give researchers unique opportunities to carry out scientic studies. As of 2013 XSEDE consists of 16 systems with varied architectural designs and capabilities. The hardware heterogeneity and software diversity make e• cient utilization of such a federation of computing resources very challenging. For example, users are constantly faced with a myriad of possibilities to build and run an application: compilers, numerical libraries, and runtime parameters. In this paper we report performance data of several popular scientic applications built with di• erent compilers and numerical libraries available on two XSEDE systems: Kraken and Gordon, and suggest the best way to compile applications for optimal performance. By comparison, we validate SU conversion factors between the aforementioned XSEDE systems from applications viewpoint.

Categories and Subject Descriptors

H.3.4 [**Performance evaluation (e• ciency and e• ectiveness)**]:

General Terms

Performance

Keywords

Performance; Benchmarking; Compilers; Numerical Libraries

1. INTRODUCTION

XSEDE is a leading distributed cyberinfrastructure for open scientic research in the United States. Users new to XSEDE, especially those at allocation proposal writing stage, are often overwhelmed by the large number of resources at their choice, the allocation policies, and the SU conversion factors. XSEDE consists of sixteen systems with di• erent designs and capabilities: High Performance Computing (HPC), which emphasizes oating-point computation, High Throughput Computing (HTC), which lends itself to executing loosely coupled distributed applications on idle computers opportunistically, and Visualization, which enables remote interactive visualization and data analysis. As the workhorse of XSEDE, the HPC platforms range from traditional CPU centric (Kraken, Lonestar) and large coherent shared memory (Blacklight) to data intensive (Gordon, Trestles) and GPU/accelerator-assisted (Keeneland, Stampede) PC clusters.

XSEDE jobs are formally charged in XSEDE Service Units (XD SUs). In light of the di• erences in the computing speed, an SU conversion rule among resources is implemented to level the playing eld. A Local SU is dened as one core-hour on a platform, and an XD SU is one core-hour on a TeraGrid Phase-1 Distributed TeraScale Facility (DTF) platform, which serves as the gold standard and was a 1.3-GHz Intel Itanium2-based cluster in 2001. The conversion rate between SUs (Local to Local, and Local to XD) is the performance ratio of the core-count-normalized High Performance LINPACK (HPL) benchmark result on both systems using *all* of available computing resources. See Table 1.

System	XD SUs
Blacklight	1.8
Gordon	4.93
Keeneland	34.0
Kraken	2.04
Lonestar	2.09
Longhorn	1.94
Nautilus	1.57
Stampede	4.6
Steele	1.61
Trestles	2.3

Table 1: Local SU to XD SU Conversion as of February 2013.

However, there are two issues with the current practice of SU conversion. First, in pursuit of fastest possible oating-point computation, HPL ignores storage capacity, bandwidth, and memory capacity, which are equally important for a balanced system. HPL has also been criticized for being a misleading metric because it rarely reects the actual achievable sustained performance of *real* scientic application workloads or makes the best use of the latest hardware technologies such as GPUs & coprocessors [12]. From a practical point of view, if a system does not submit HPL result to TOP500, such as NCSA Blue Waters, its SU conversion calculation would be a problem. Secondly, the HPL result is based on full system run, but very few XD users are able to use even half of a system [10]. In addition, the o• cial SU conversion rule does not take issues such as application scalability and problem size into consideration[1].

Aggravating the complexity of hardware heterogeneity is the plenitude of software and tools available to achieve the same goal. Take the routine task of building a scientic application as an example, the XSEDE software stack provides four compiler avors: the standard GNU compilers supplied with Linux, the Intel compiler suite, the Portland Group (PGI) compilers, and the Cray compilers (only available on Kraken). If an application uses linear algebra libraries such as BLAS or LAPACK, one has hardware-optimized Intel MKL (Math Kernel Library), ACML (AMD Core Math Library), ATLAS (Automatically Tuned Linear Algebra Software) [14], GotoBLAS, and GotoBLAS-derived Cray LibSci (only available on Kraken) to choose from. Standard fast Fourier transform APIs add another dimension of alternatives, as there are FFTW2 and FFTW3 [7]. Each of the factors has further ramications: di • erent versions, tunable ags, and runtime parameters such as serial or threaded, processor a• nity, and NUMA/data placement. An exhaustive search of all possible build and run options to achieve the optimal application performance on every XSEDE resource could be daunting and could consume considerable allocation SUs. In reality, users and supercomputer support sta• can only rely on anecdotal experience or word of mouth, such as using Intel compilers & libraries on Intel processors and PGI compiler & ACML on AMD processors, but a thorough study is lacking.

This paper attempts to address the above questions. We report comprehensive performance measurements of popular parallel HPC applications [8] built with three compilers and numerical libraries on two XSEDE platforms. The cross-platform comparison of the application performance enables us to derive more realistic, *application-based* SU conversion factors. This work extends our prior PEAK project [9] which focuses on benchmarking seven key routines of BLAS & LAPACK libraries from three vendors in the single-node environment.

A related e• ort in this area is SPEC MPI 2007 [13], a commercial benchmark suite comprising 18 MPI-parallel oating-point intensive scientic codes and kernels. It also hosts a public repository of submitted data. However, obtaining the benchmark software and publishing the results both require hefty fees. The available results are sporadic in terms of build options and MPI rank sizes, and its application selection or tested machines do not necessarily align closely with the interests of XSEDE community.

The rest of the paper is organized as follows. In section 2 we describe the experiment setup, including the compilers and libraries, the benchmark codes, and the hardware systems. Section 3 presents the our experimental evaluation and interprets the benchmark results. We discuss the SU conversions based on these results in Section 4 and conclude our work and future directions in Section 5.

2. EXPERIMENTAL ENVIRONMENT

2.1 Hardware

We conducted experiments on Kraken and Gordon. We chose them because they have representative distinct features, such as di• erent processor vendors (AMD vs. Intel) and di• erent interconnects (proprietary vs. commodity.) In general, after normalization Gordon is at least twice as powerful and e• cient as Kraken, as in Table 3.

2.1.1 Kraken

Kraken is a Cray XT5 supercomputer managed by the National Institute for Computational Sciences (NICS) located at the Oak Ridge National Laboratory. Kraken has a total of 9,408 compute nodes, each of which has two 2.6 GHz hexa-core AMD Opteron 2435 processors. There are 112,896 cores in total with a peak performance of 1.17 PetaFLOPS. Each compute node has 16 GB memory and is attached to a Lustre parallel le system. Krakens high-speed low-latency network is made of Cray SeaStar2+ chips on board plus a scalable 3D torus interconnect fabric.

2.1.2 Gordon

Located at San Diego Supercomputer Center, Gordon is an 1024-node large-memory PC cluster designed for data-intensive jobs. Each compute node has two 2.6 GHz octa-core Intel Xeon E5-2670 processors, 64 GB memory, and 80 GB solid-state disk. There are 16,384 cores in total with a peak performance of 341 TeraFLOPS. Gordon also uses Lustre le system and has a dual-rail 3D torus InniBand QDR (40 Gbit/s) network.

2.2 Application Build Environment

We used the following compilers, BLAS & LAPACK libraries, and FFTW libraries.

On Kraken, the compiler options are: GNU compiler collection version 4.6.2, Intel compiler 12.1.2, PGI compiler 11.9.0. All of these compiler suites support C, C++, and Fortran. The BLAS & LAPACK implementations are ACML version 4.4.0, Cray LibSci 11.0.4, Intel MKL version 11.1.038. The FFTW libraries are versions 2.1.5.3 and 3.3. The MPI library is Cray MPT version 5.3.5.

On Gordon, the compiler options are: GNU 4.6.1, Intel 12.1.0, and PGI 12.8.1. The BLAS & LAPACK implementations are ACML version 5.3 and Intel MKL version 10.3.7. The FFTW libraries are versions 2.1.5 and 3.3. The MPI library is MVAPICH2 version 1.8a1p1.

It should be noted that FFTW2 and FFTW3 are two incompatible application programming interfaces [7]. FFTW works in two stages: Planning, in which FFTW auto-tunes itself by adapting to the hardware, and Execution, which is the actual computation. In FFTW2, once the one-time planning is done, the result is applied to any arrays and

[1]XSEDE does accept custom SU transfer rules using requesters own benchmark results instead of the default HPL criteria, but this needs special request.

any multiplicity/stride parameters. FFTW3 shifts the Planning/Execution division of labor more towards Planning, so the planner now also adapts to the input problem and parameters. Understandably, not all benchmark codes in our test can support both FFTW2 and FFTW3.

2.3 Benchmark Codes

We selected popular scientic applications based on Hadris report [8] , source code availability, and license agreement[2]. All of them happened to be Molecular Dynamics (MD) [11] simulation codes. Their main di• erences lie in the force eld of choice, long-range potential computation algorithm, neighbor list cuto• radius, and temperature control scheme.

Amber [1] (Assisted Model Building Energy Renement) is a suite of MD codes designed specically for biomolecular systems. We used Amber version 12 and its PMEMD (particle mesh Ewald for MD) program in our benchmarking. PMEMD is written in Fortran 90 and parallelized using a master/slave data replication programming model.

Gromacs [2] (GROningen MAchine for Chemical Simulations) is another MD code written in C/C++ with emphasis on algorithmic and processor-specic optimization. Its non-bonded potential computational kernels contain manually tuned assembly code and therefore is able to achieve the fastest single CPU performance compared to other competing MD programs. We used Gromacs version 4.5.3.

LAMMPS [3] (Large-scale Atomic/Molecular Massively Parallel Simulator) is a C++ based MD code. It features a highly modular and extensible design and 80% of its code base is contributed by developers and users all over the world. We used October 2012 version of LAMMPS.

NAMD [4] (Not Another MD) is a peta-scale MD code written using the object-oriented Charm++ parallel programming model. NAMD implements load balancing and can assign certain computation to any processor, thus achieving ultra scalability to the tune of 300,000 processor cores on a 100-million-atom input problem. We used NAMD version 2.9

Written in Fortran, CPMD [5] (Car-Parrinello MD) is a plane wave/pseudo-potential implementation of Density Functional Theory for *ab-initio* MD. That is, unlike previous software packages which use phenomenological interatomic and intermolecular potentials (classical MD), CPMD calculates the potential energy surface directly from the rst quantum chemistry principles. Therefore, it can model unusual bond breaking/making and electronic structure changes. Its drawback is the problem size is limited to hundreds of atoms. We used CPMD version 3.15.3.

2.4 Compiling and Executing Framework

Building modern HPC applications using di• erent compilers presents a great challenge. This is especially true for C++ and Fortran 9x, which have complex feature sets and rich non-standard extensions. In the course of this research, for example, we have experienced compiler crash when compiling certain source le using specic optimization ags, or compilation error because the new version of the compiler complains certain syntax as error while the old version compiles just ne.

As in our previous work [9], for each application we wrote Python scripts to automate the build process using di• erent combinations of compilers and numerical libraries. We also had scripts to automatically create and submit batch job scripts, collect data, and plot results.

On both Kraken and Gordon, an application-build script generates a series of shell scripts from a given list of compilers and libraries and execute them. A typical shell script rst cleans up the default programming environment via `module unload` commands, loads the target compilers and libraries, sets up necessary environment variables (e.g. `CC`, `F77`, `CFLAGS`, `MPICC`), and executes applications own `configure` script or tweak the makeles, followed by `make` command invocation. The detailed `configure` script options and build ags for each benchmark code can be found in Table 2.

Amber
Kraken: ./congure --no-updates -static -crayxt5 -mpi (gnu/intel/pgi) Gordon: ./congure --no-updates -mpi (gnu/intel /pgi)

CPMD
GNU: export FFLAGS=-O2 -fopenmp -fcray-pointer export CPPFLAGS=-D __Linux -D__GNU -DFFT_(FFTW/FFTW3) -DPOINTER8 -DPARALLEL=parallel Intel: export FFLAGS=-O2 -openmp -mkl export CPPFLAGS=-D __Linux -DFFT_(FFTW/FFTW3) -DPOINTER8 -DPARALLEL=parallel PGI: export FFLAGS=-O2 -mp export CPPFLAGS=-D __Linux -D__PGI -DFFT_(FFTW/FFTW3) -DPOINTER8 -DPARALLEL=parallel -D__DERF

Gromacs
./congure --with-• t=• tw(2/3) --enable-mpi --enable-double --without-qmmm-gaussian --without-x --without-xml --disable-shared

LAMMPS
export FFT_INC=-DFFT _FFTW(2/3) export FFT_LIB=-l(• tw/• tw3)

NAMD
./build charm++ mpi-linux-x86_64 --no-build-shared -O3 -DCMK_OPTIMIZE=1

Table 2: Important Flags Used to Build Benchmark Codes

The job submission and result collection are also streamlined. When the runs are completed, the data are gathered to plot the performance and are stored in a database. We aim to build a comprehensive knowledge base containing all performance data of important HPC applications built with varieties of compilers and numerical libraries on all XSEDE platforms. The results would eventually be made available on the XSEDE Metrics on Demand (XDMoD) website [6].

3. EXPERIMENT RESULTS

In our test, each build version of an application is executed three times and the shortest execution time is selected for the charting and analysis. The benchmark is done in the *strong scaling* manner, that is, the input problem and parameters stay xed but the number of cores varies. As later results will demonstrate, when the core count grows, the

[2]Certain most-used codes such as VASP are not open-source and have restrictive user license agreements, so we do not include them in this study.

performance di• erence due to compilers or libraries usually diminishes. This should not be surprising because as the number of cores goes up, the amount of per-core work decreases and the serial part and inter-process communication overhead starts to dominate.

In the following paragraphs we describe for each application, the compilers and libraries used, the input dataset, and the result analysis. For consistency, in all plots we use xed colors for compiler vendors: green for PGI, red for GNU, and blue for Intel. Regarding the FFTW libraries, we distinguish by point types: triangle for FFTW2 and circle for FFTW3.

3.1 Amber

We built Ambers PMEMD using all three compiler a-vors. PMEMD uses FFTW3 but not BLAS & LAPACK. We tested the Joint Amber/Charmm (JAC) benchmark input: 23,558 atoms, explicit TIP3P solvent PME, 1 fs step size, 1,000 steps, 9 Å direct space cuto• , NFFT=64 • 64 • 64, shake_tol = $10^{\cdot\,7}$, NVE, and ntpr=100. Figure 1 shows the results. On Kraken the Intel version does not run, so there is no result for it.

On Gordon, the PGI version falls behind GNU and Intel versions by up to 30%, and the Intel version has a slight performance advantage over the GNU version. Ambers scalability on JAC benchmark starts to worsen as the core count grows beyond 64 (4 nodes) so we do not include that part in the plot. The case for Kraken is more interesting. On Kraken the PGI version runs faster when core count is 12 and 24. The performance deteriorate signicantly when we use 48 cores (4 nodes) or more.

3.2 CPMD

CPMD is a typical application that can be optimized by many build options: compiler, BLAS/LAPACK library, and FFTW version. Theoretically we are able to produce 3 • 3 • 2 = 18 versions of CPMD on Kraken. However, the latest source code version of CPMD is not compatible with the version of GNU compiler on Kraken, we thus only produced 12 builds of CPMD on Kraken. We drove CPMD simulation with input problem Si_{512}: Local-Density Approximation, Kleinman pseudo-potential, 20 Ry wavefunction cuto• , • 320K plane waves, 108 • 108 • 108 real space mesh.

Due to memory requirement, the minimal core count to run Si_{512} benchmark on Kraken is 48. From the benchmark results in Figure 2, it is obvious that on Kraken, the choice of compiler has substantial inuence on the performance. Overall, the PGI compiler produces much faster CPMD executables. In most cases, the Intel versions run three to six times slower than their PGI counterparts. The only two exceptions are intel-libsci-• tw2 and intel-libsci-• tw3, which have very similar performance to PGI versions. The use of FFTW library does not seem to have signicant impact.

On Gordon, the inuence of compiler become smaller. On the other hand, the choice of numerical library does matter: the MKL versions outperform ACML versions to a great extent. The di• erence caused by numerical libraries diminish when more cores are used. As we can see from the gure, intel-mkl-• tw3 is the best combination that we can use to build CPMD on Gordon.

3.3 Gromacs

Gromacss main simulation program mdrun supports

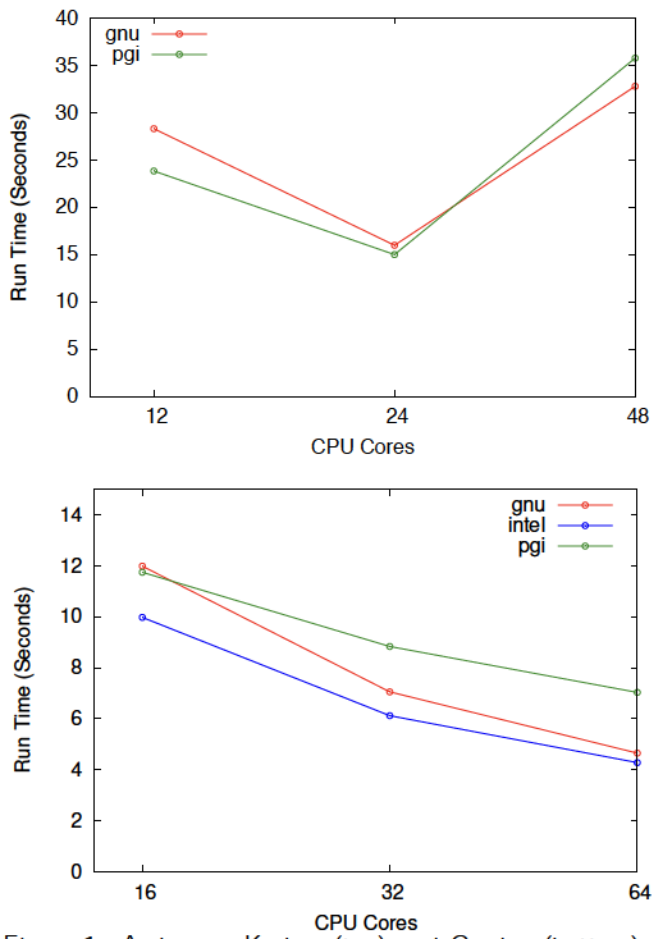


Figure 1: Amber on Kraken (top) and Gordon (bottom) Performance.

a variety of FFT libraries and we built for FFTW2 and FFTW3. Regarding BLAS/LAPACK, Gromacs only makes use of them in its utility programs but not in mdrun, so we omit them. The benchmark input is d.dppc: 1,024 DPPC lipids with 23 water molecules per lipid (121,856 atoms in total), twin-range group based cuto• : 1.8 nm for electrostatics and 1.0 nm for Lennard-Jones interactions, long-range contribution to electrostatics updated every 10 steps, and 5,000 steps (10 ps step size)

The results are shown in Figure 3. Overall, PGI builds are the worst on both systems, lagging behind Intel and GNU by 6-11% on Kraken and 10-18% on Gordon. Regarding FFTW2 and FFTW3, intel-• tw3 is the fastest build combination on Kraken, but there is no clear winner on Gordon.

3.4 LAMMPS

We compiled six versions of LAMMPS, one for each compiler and FFTW combination. LAMMPS does not need BLAS/LAPACK. The benchmark problem is Rhodo: rhodopsin protein in solvated lipid bilayer, CHARMM force eld with a 10Å cuto• Lennard-Jones cuto• (440 neighbors per atom), particle-particle particle-mesh (PPPM) for long-range Coulombics, and NPT integration.

Figure 4 shows the result. Generally, the e• ect due to FFTW libraries is almost negligible on Gordon, while on Kraken, they have certain impact when 48 cores are used.

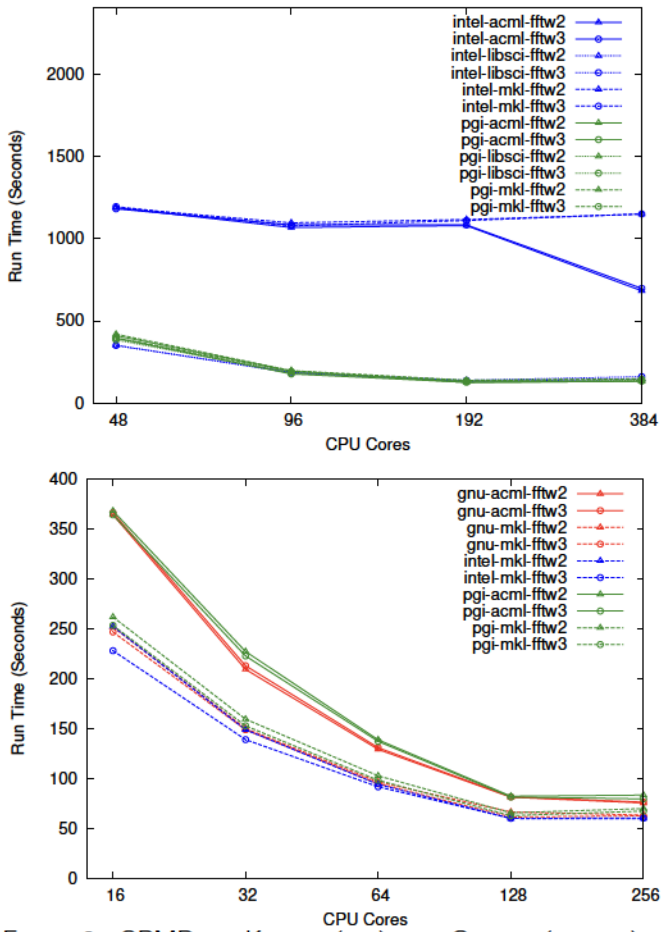


Figure 2: CPMD on Kraken (top) and Gordon (bottom) Performance

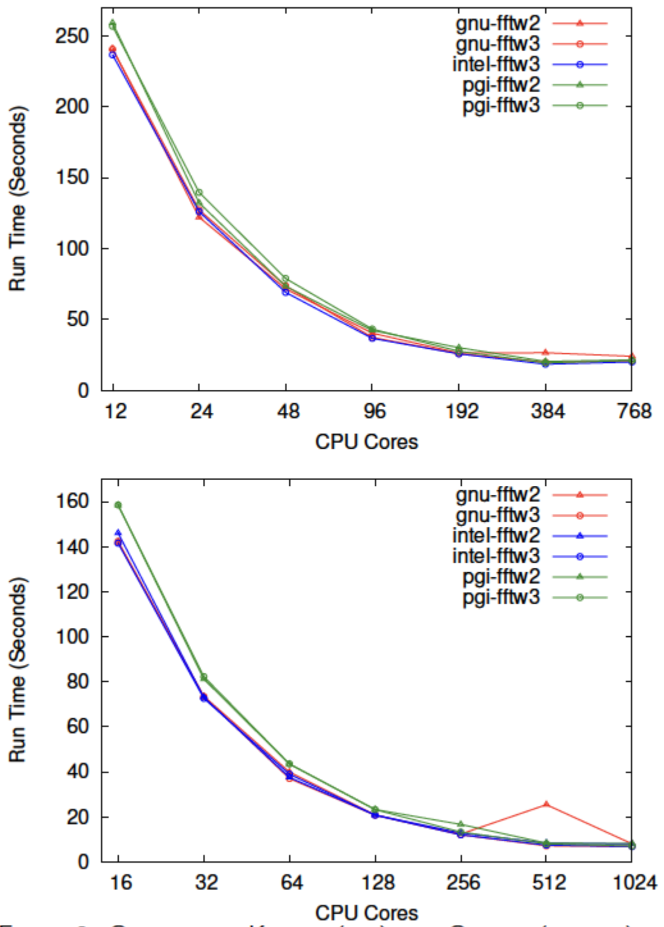


Figure 3: Gromacs on Kraken (top) and Gordon (bottom) Performance

The GNU and Intel versions have very similar performance, while the Intel version is only slightly faster than its GNU counterpart on Kraken. The PGI version, on the other hand, is slower by a large margin of 8-16%.

3.5 NAMD

NAMD is built merely for each compiler avor because the sole numerical library NAMD uses is single-precision FFTW2 (version 2.1.5). The input dataset is a one-million-atom large molecular system, Satellite Tobacco Mosaic Virus (STMV), and the simulation parameters are: 216• 216• 216 PME grid size, 12Å cuto• , PME every 4 steps, periodic, and 500 steps.

Figure 5 shows the result. On Kraken, the PGI build is not able to run, so there are no results for this conguration. The GNU version exhibits 2-8% performance advantage over Intel . On Gordon, the PGI version fails to produce expected result when core counts are 256 and 512. Overall, Intel edges out GNU and trumps PGI by 15%.

4. SU CONVERSION AND CONSUMPTION ANALYSIS

The experimental results in previous sections enable us to validate the present XSEDE SU conversion against the actual cross-platform application performance. As mentioned in Section 1, the SU conversion is based on the core-wise High Performance LINPACK (HPL) benchmark result[3]. The current o• cial SU conversion rate stands at one Gordon SU equal 2.42 Kraken SUs (=4.93/2.04. See Table 3.) We suspect this conversion is anachronistic and is based on an outdated Kraken conguration, which achieved 7.02 GFLOPS per core[4], against a modern Gordon, which scored 17.69 GFLOPS per core[5]. Kraken has since undergone major hardware upgrades in late 2009 (faster processors) and early 2011 (more nodes) and attained 8.14 GFLOPS per core, 16.5% speedier than the old Kraken. We believe a more sensible HPL-based SU conversion rate should be 17.69/8.14 = 2.17. In other words, if XSEDE SU is fungible like money, then Kraken has been underpriced. In Table 3, we list ratios computed with known specs of Kraken and Gordon.

For application-oriented SU conversion and performance comparison, we need to account for di• erent run sizes on Kraken and Gordon (multiples of 12 vs. 16). We use the following formula:

$$\text{Gordon/Kraken Ratio} = \frac{\text{Kraken SUs consumed}}{\text{Gordon SUs consumed}}$$

[3]Available at http://www.top500.org
[4]463 TFLOPS on 66,000 cores. See June 2009s Top500 list.
[5]286 TFLOPS on 16,160 cores. See November 2012s Top500 list.

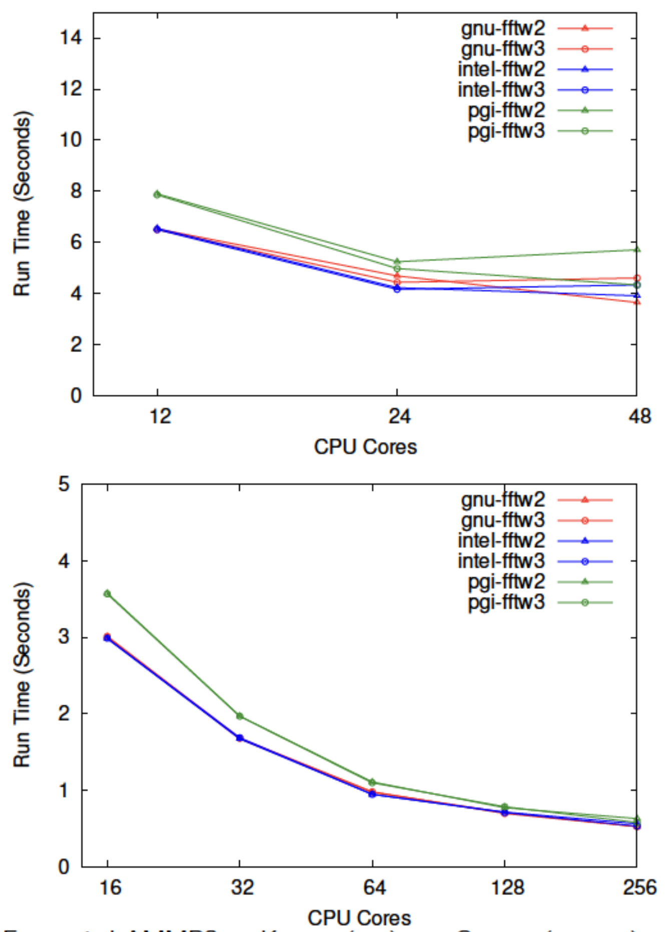


Figure 4: LAMMPS on Kraken (top) and Gordon (bottom) Performance.

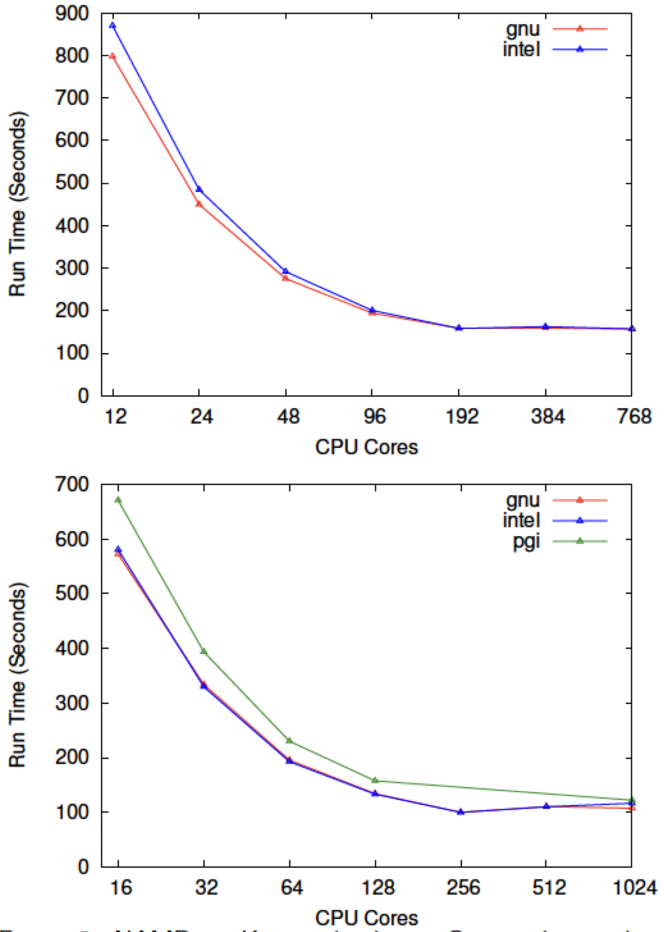


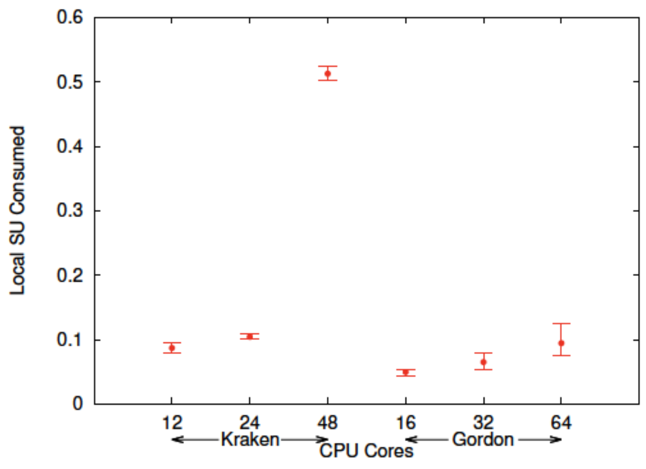
Figure 5: NAMD on Kraken (top) and Gordon (bottom) Performance.

For comparable run sizes (e.g., 12 and 16 (one node), 24 and 32 (2 nodes), etc.), we calculate ratios: SU consumed by a Krakens run vs. SU consumed by a Gordons run, for all compiler/libraries builds. If the application-based SU conversion rate drops below the o• cial rate of 2.42, it suggests that Kraken is the cheaper platform to run that application because it costs less XD SUs. From Figure 6f we can see that Kraken is the preferable platform to run Gromacs and NAMD. For NAMD, regardless of compilers used, its SU conversion rate is never greater than 1.2. This means, although it takes 20% more time to run NAMD on Kraken than on Gordon for similar run sizes, Gordon nevertheless charges more than twice of XD SUs.

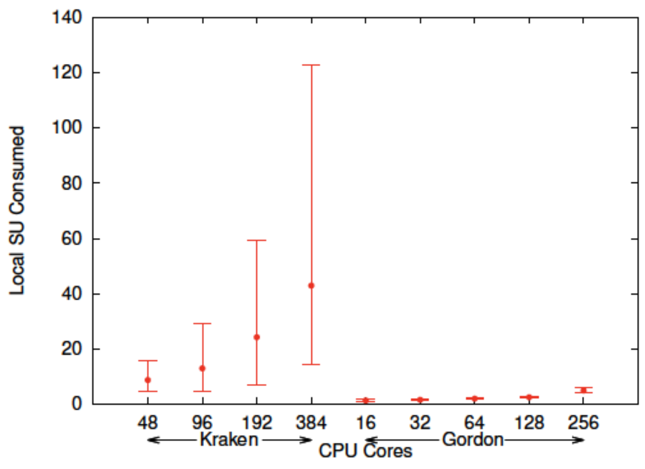
We calculated the 95% condence intervals of the mean SU conversion rates for each applications and the average, see Figure 6f and Table 4. Moreover, we gathered performance proles of each run on Kraken. In Table 4, we also list average percentage of communication time out of total walltime on Kraken. Here are interesting observations: rst, there is a strong correlation between Gordon-Krakens SU conversion rate and communication time of an application. The SU conversion rate is lower when an application spends less time on communication. Second, we nd a very big range for the condence interval of Amber SU conversion rate. This is due to the performance di• erences on Kraken and Gordon for certain number of nodes. Notice from Figure 1 that Ambers performance drops from using 2 nodes

Per Core	**Kraken**	**Gordon**
CPU	2.6GHz AMD Opteron 2435	2.6GHz Intel Xeon E5-2670
Memory Interface	DDR2-800	DDR3-1333
Peak GFLOPS	10.4	20.8
HPL GFLOPS	8.14	17.69
L3 Cache (MB)	1	2.5
Memory Size (GB)	1.33	4
Memory Bdwth (GB/s)	2.13	5.33
Peak MFLOPS/Watt	380.02	937.62
HPL MFLOPS/Watt	297.44	797.43
Comparison	**Gordon/Kraken Ratio**	
Peak FLOPS	2	
HPL FLOPS	2.17	
Memory Bandwidth	2.5	
Peak MFLOPS/Watt	2.46	
HPL MFLOPS/Watt	2.68	
O• cial SU Conv. Factor	**2.42**	

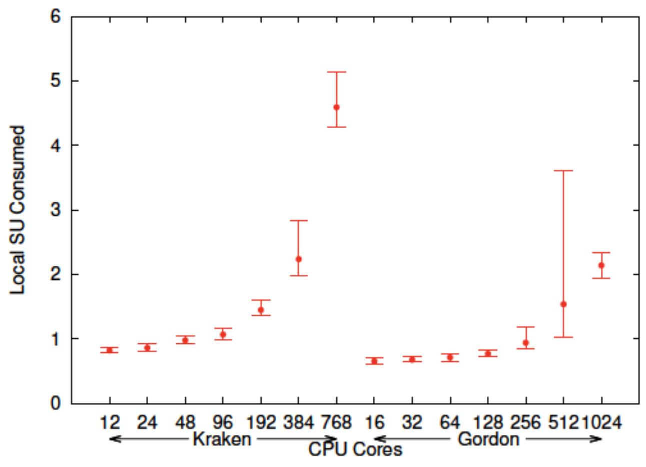
Table 3: Kraken and Gordon Comparison. The HPL result is from November 2012s TOP500 list.



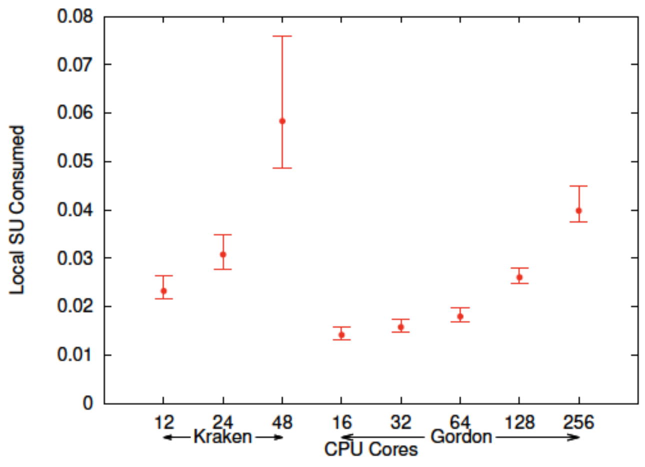
(a) Amber SU Consumption.



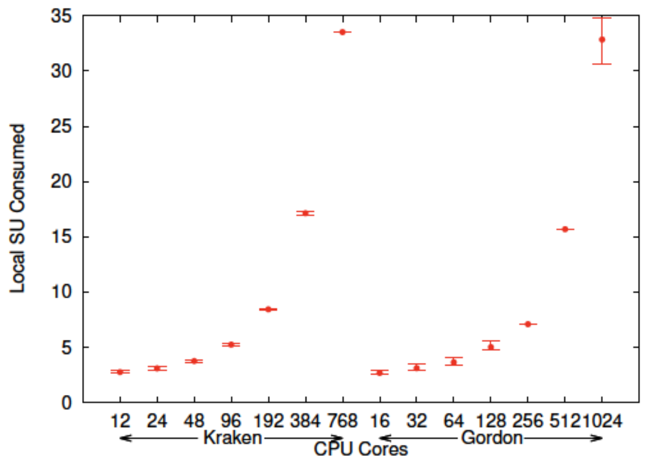
(b) CPMD SU Consumption.



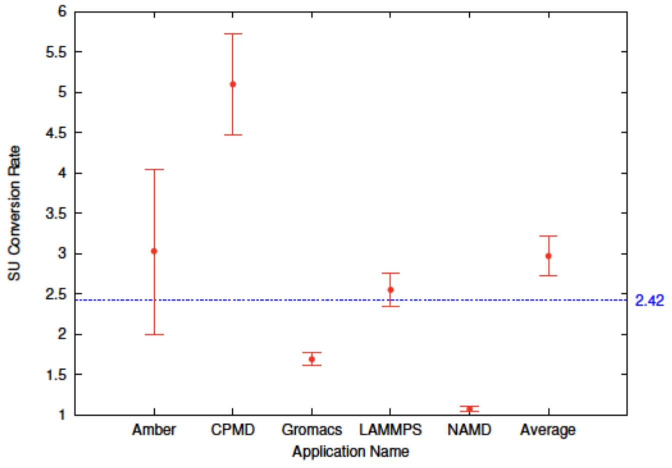
(c) Gromacs SU Consumption.



(d) LAMMPS SU Consumption.



(e) NAMD SU Consumption.



(f) SU Conversion Condence Intervals

Figure 6: SU consumption comparison and conversion rates

to 4 nodes (48 cores) on Kraken, while the counterpart of 4 nodes (64 cores) running on Gordon scales well. This phenomenon leads to the ill-behaved SU conversion rate for the 4 nodes situation, see Table 5. Therefore, in order to achieve the best performance, the number of nodes requested on di• erent supercomputers varies even for the same scientic

Applications	Application-Based SU Conversion Rate	95% Condence Interval	Average Communication
Amber	3.03	(2.00, 4.05)	84%
CPMD	5.10	(4.47, 5.73)	62%
Gromacs	1.69	(1.61, 1.77)	28%
LAMMPS	2.55	(2.34, 2.76)	74%
NAMD	1.07	(1.04, 1.11)	3%

Table 4: The application-based SU conversion rates between Kraken and Gordon for ve applications.

applications.

Applications	SU Conversion Rate
Amber 4 nodes	5.68
Amber 1 & 2 nodes	1.70

Table 5: The SU conversion rates for Amber.

Third, based on the statistical inference on the benchmark testing data, we are 95% condence that the application-based SU conversion rate is between 2.72 and 3.22.

A closely related topic concerns the SU consumption. An XSEDE allocation proposal must specify the amount of XD SUs required, and it is imperative that users are mindful of their SU burn rates and ensure their allocation is not depleted too fast. Therefore, we examine how many SUs are consumed for our benchmark codes. Figure 6a-6e show the SU consumption for the best, mean, and worst run time (among all compiler/libraries builds) of Amber, CPMD, Gromacs, LAMMPS and NAMD. If a parallel code has perfect scalability (in the *strong scaling* sense), its consumed SUs should remain at without regard to the core count because SU represents the amount of work done. All of the three codes shown here have increasing SU consumption in proportion to the number of assigned cores. We theorize two possible causes: either the code itself indeed has considerable parallel overhead, or there are environmental e• ects such as interconnect tra• c and node distance and topology. Our result also points out that the best way to preserve SUs while getting computation done is to use the smallest run size.

5. CONCLUSION AND FUTURE WORK

In this paper we report performance analysis of popular molecular dynamics codes compiled with various compilers and numerical libraries and executed in di• erent modes on Kraken and Gordon XSEDE systems. Our results shed light on the best way to compile these codes for optimal performance. We also derive the application-based SU conversion factors and compare them against the o• cial SU conversion factors. We nd that Kraken is more cost-e• ective to run molecular dynamics applications such as NAMD and Gromacs.

In the future we would also like to assess the impact of MPI implementations (Intel MPI, MVAPICH1/2, and Open MPI) and topology-aware process placement strategies on application performance. We will extend our work to all XSEDE platforms, with higher priority on heterogeneous systems such as Keeneland (NVidia CUDA) and Stampede (Intel Xeon Phi/MIC) because they have received much attention regarding their usability and power/performance efciency. We will also cull applications from broader elds of science and cover all of Phil Colellas parallel dwarfs, e.g. computational chemistry, weather/climate, and data-intensive codes.

Finally, an easy-to-use charting and analysis tools for our benchmark data is under construction and will be part of XSEDE Metrics on Demand (XDMoD) website [6].

Acknowledgment

This research used resources at the National Institute for Computational Sciences supported by the National Science Foundation. This research was also supported in part by the National Science Foundation under award number 1025159.

6. REFERENCES

[1] http://ambermd.org.

[2] http://www.gromacs.org.

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

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

[5] http://www.cpmd.org.

[6] http://xdmod.ccr.buffalo.edu.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005.

[8] B. Hadri, M. Fahey, T. Robinson, and W. Renaud. Software usage on Cray systems across three centers (NICS, ORNL and CSCS). In *Cray User Group Conference (CUG)*, 2012.

[9] B. Hadri, H. You, and S. Moore. Achieve better performance with PEAK on XSEDE resources. In *First XSEDE Conference Proceedings*, 2012.

[10] D. L. Hart. Measuring TeraGrid: Workload characterization for an HPC federation. *International Journal of High Performance Computing Applications*, 25(4):451–465, 2011.

[11] J. Hein and et al. On the performance of molecular dynamics applications on current high-end systems. *Philosophical Trans. of Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1833):1987–1998, 2005.

[12] W. Kramer. Top problems with the TOP500. http://www.ncsa.illinois.edu/News/Stories/TOP500problem/.

[13] Standard Performance Evaluation Corporation. SPEC MPI 2007. http://www.spec.org/mpi/.

[14] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.