Performance Tuning of Fock Matrix and Two-Electron Integral Calculations for NWChem on Leading HPC Platforms

Hongzhang Shan, Brian Austin, Wibe De Jong, Leonid Oliker, Nicholas Wright
CRD and NERSC
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
{hshan, baustin, wadejong, loliker, njwright}@lbl.gov

Edoardo Apra
WR Wiley Environmental Molecular Sciences Laboratory
Pacific Northwest National Laboratory, Richland, WA 99352
edoardo.apra@pnnl.gov

***Abstract*—Attaining performance in the evaluation of two-electron repulsion integrals and constructing the Fock matrix is of considerable importance to the computational chemistry community. Due to its numerical complexity improving the performance behavior across a variety of leading supercomputing platforms is an increasing challenge due to the significant diversity in high-performance computing architectures. In this paper, we present our successful tuning methodology for these important numerical methods on the Cray XE6, the Cray XC30, the IBM BG/Q, as well as the Intel Xeon Phi. Our optimization schemes leverage key architectural features including vectorization and simultaneous multithreading, and results in speedups of up to 2.5x compared with the original implementation.**

I. Introduction

NWChem [21] is an open source computational chemistry package for solving challenging chemical and biological problems using large scale ab initio molecular simulations. Since its open-source release three years ago, NWChem has been downloaded over 55,000 times world wide and played an important role in solving a wide range of complex scientific problems. The goal of NWChem is to not only provide its users a computational chemistry software suite, but also to provide fast time to solution on major high-performance computing (HPC) platforms. It is essential for a software tools like NWChem to effectively utilize a broad variety of supercomputing platforms in order for scientists to tackle increasingly larger and more complex problem configurations.

Moden HPC platforms represent diverse sets of architectural configurations, as clearly seen in different processor technologies and custom interconnects of the current top five fastest computers in the world [20]. The No.1 system employes Xeon E5-2682 and Xeon Phi (Intel MIC) processors while the No. 2 system uses Opteron 6274 and Tesla K20X processors. These two platforms follow similar host+accelerator design patterns. The No. 3 and 5 systems use IBM PowerPC A2 processors (IBM BG/Q), and No. 4 uses SPARC64 processors. The diversity of the high performance computing architectures poses a great challenge for NWChem to make efficient use of these designs.

In this work, we address this issue by tuning the performance of NWChem on three currently deployed HPC platforms, the Cray XE6, the Cray XC30, the IBM BG/Q, and the Intel MIC. Although NWChem provides extensive capabilities for large scale chemical simulations, we focus on one critical component: the effective evaluation of two-electron repulsion integrals with the TEXAS integral module that are needed for the construction of the Fock matrix needed in the Hartree-Fock (HF) and Density Functional Theory calculations [7]. This capability consists of about 70k lines of code, while the entire NWChem distribution includes more than 3.5 million lines. Figure 1 shows the total running times of our NWChem benchmark c20h42.nw on the 24-core node of Hopper, the Cray XE6 platform. This benchmark is designed to measure the performance of the Hartree-Fock calculations on a single node with a reasonable runtime for tuning purposes. Each worker is a thread created by the Global Array Toolkit [4], which is responsible for the parallel communication within NWChem. The total running times are dominated by the Fock matrix construction and primarily the two-electron integral evaluation.

By exploring the performance on these four platforms, our goal is to answer the following questions and infer guidelines for future performance tuning. *(I)* What kind of optimizations are needed? *(II)* How effective are these optimizations on each specific platforms? *(III)* How should the code be adapted to take maximum advantage of the specific hardware features, such as vectorization and simultaneous multithreading? *(IV)* Is there a consistent approach to optimize the performance across all four platforms? Our insights to these questions are discussed and summarized in Sections VIII and X.

The rest of the paper is organized as follows. In Section II, we describe the four experimental platforms and in Section III we briefly introduce the background for Fock matrix construc-

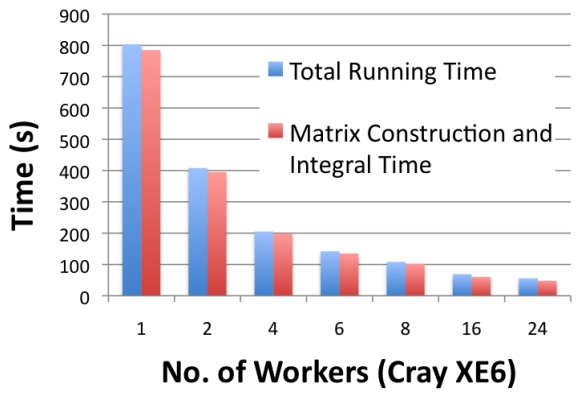

Fig. 1: The total running times and the corresponding fock matrix construction and integral calculation times on Hopper.

tion and two electron repulsion integral calculations. The tunning processes and the corresponding performance improvements on the four platforms are discussed in Sections IV, V, VI, VII individually. We summarize the tuning results in Section VIII. Related work is discussed in Section IX. Finally we summarize our conclusions and outline future work in Section X.

II. Experimental Platforms

To evaluate the impact of our optimizations, we conduct experiments on four leading platforms, the Cray XE6, the Cray XC30, the Intel MIC, and the IBM BG/Q. Below, we briefly describe the four platforms we used and Table I summarizes the important node characteristics of these four different architectures .

A. The Cray XE6, Hopper

The Cray XE6 platform, called Hopper [10], is located at NERSC and consists of 6,384 dual-socket nodes connected by a Cray Gemini high-speed network. Each socket within a node contains an AMD "Magny-Cours" processor with 12 cores running at 2.1 GHz. Each Magny-Cours package is itself a MCM (Multi-Chip Module) containing two hex-core dies connected via HyperTransport. Each die has its own memory controller that is connected to two 4-GB DIMMS. This means each node can effectively be viewed as having four chips and there are large potential performance penalties for crossing the NUMA domains. Each node has 32GB DDR3-800 memory. The peak data memory bandwidth is 51.2GB/s. Each core has 64KB L1 cache and 512KB L2 cache. The six cores on the same die share 6MB L3 cache. The compiler is PGI Fortran 64-bit compiler version 13.3-0.

B. The Cray XC30, Edison

The Cray XC30 platform, called Edison [2], is also located at NERSC. The first stage of the machine contains 664 compute nodes connected together by the custom Cray Aries high-speed interconnect. Each node is configured with 64GB DDR3-1666 memory. The peak memory bandwidth is 106 GB/s. Each node has two 8-core intel Sandy Bridge processors (16 cores total) running at the speed of 2.6 GHz. Each core supports two hyper threads and supports 4-way SIMD vector computations. Each core has 64KB L1 cache and 256KB L2 cache. The 8-core processor has 20MB L3 cache shared by all eight cores. The compiler we used is Intel Fortran 64 Compiler XE version 13.0.1.

C. The Intel MIC, Babbage

The Intel MIC platform, called Babbage [15], is a new test platform located at NERSC. There are total 45 compute nodes connected by the Infiniband interconnect. Each node contains two MIC cards and two Intel Xeon host processors. Each MIC card contains 60 cores running at the speed of 1.05 GHz and 8 GB GDDR5-2500 memory. The peak memory bandwidth is 352GB/s. However, using STREAM benchmark, the best achievable bandwidth is only around 170GB/s. Each core supports 4 hardware threads and an 8-way SIMD vector processing unit capable of delivering 16.8 GF/s floating-point computations. Each core has 32KB L1 cache and 512KB L2 cache. The compiler we used is Intel Fortran 64 Compiler XE version 13.0.1.

D. The IBM BG/Q, Mira

The IBM BG/Q platform, called Mira [16], is located at Argonne National Laboratory. Each node contains 16 compute cores (IBM Power A2) and 1 supplemental core to handle operating system tasks. The memory size on a node is 16 GB DDR3-1333 with 42.6 GB/s peak memory bandwidth. Each core supports 4-way simultaneous multithreading and a QUAD SIMD floating point processing unit capable of 12.8 GF/s computing speed. Each core has 16KB L1 data cache and all 16 cores share the 32MB L2 cache. The memory on the node follows SMP (symmetric multiprocessing) not NUMA (nonuniform memory access) architecture. The compiler is IBM XL fortran compiler for BG/Q version 14.1.

III. Background for Fock Matrix Construction and Two-Electron Integral Evaluation

In quantum physics or chemistry, the Hartree-Fock method [7] is a fundamental approach to approximate the solution of Schrödinger's equation [5], [9], [8]. During this approach, a matrix called the Fock matrix (F) needs to be repeatedly constructed. It is a two-dimensional matrix and its dimensional size is determined by the number of basis functions, N. Elements of the Fock matrix are computed by the following formula [3]:

$$F_{ij} = h_{ij} + \sum_{k=1}^{N}\sum_{l=1}^{N} D_{kl}((ij|kl) - \frac{1}{2}(ik|jl))$$

where h is one-electron Hamiltonian, D is the one-particle density matrix, and $(ij|kl)$ is a two-electron repulsion integral. The time to construct the Fock matrix is dominated by the computation of these integrals. In NWChem, the most heavily used module to calculate the integrals is called TEXAS [22]. Given this equation, the total number of quartet integrals to

TABLE I: The Important Node Characteristics of the Four Platforms.

	Cray XE6	Cray XC30	Intel Xeon Phi (MIC)	IBM BG/Q
Node	Two 12-core AMD Opteron (24) Two Sockets, NUMA	Two 8-core Intel Xeon (16) Two Sockets, NUMA	60 Intel MICs 5110P Multi Sockets, NUMA	16 IBM Power A2 SMP
Memory	32GB DDR3 800MHz 51.2 GB/s	64GB DDR3 1666MHz 106 GB/s	8GB GDDR5 2500MHz 352 GB/s	16GB DDR3 1333MHz 42.6 GB/s
Core	2.1 GHz 1 Thread 8.4 GF/s SSE	2.6 GHz 2 Hyper Threads 20.8 GF/s 4-way SIMD (256 Bits)	1.05 GHz 4 Hyper Threads 16.8 GF/s 8-way SIMD (512 Bits)	1.6 GHz 4 Hyper Threads 12.8 GF/s Quad FU (256 Bits)
Cache	L1: 64KB, L2: 512KB L3: 6MB/ 6 Cores	L1: 32KB, L2: 256KB L3: 20MB /8 Cores	L1: 32KB, L2: 512KB	L1: 16KB, L2: 32MB (shared)

build the matrix F could reach $O(N^4)$ making it computationally prohibitive. However, by applying molecular and permutation symmetry as well as screening for small values, the complexity can be reduced to $O(N^2 - N^3)$.

The computation of each single or block of integrals is an independent operation that can be easily parallelized. However, the computational cost of each individual integral differs substantially depending on the angular momentums of the corresponding basis functions. Therefore, a dynamic load balancing method is used to ensure that all parallel workers perform equal amounts of integral computations. The Fortran pseudo-code of this approach is shown in Listing 1. The algorithm utilizes a global task counter to distribute work to a parallel worker that is available. To avoid network pressure on the global task counter, in each iteration a runtime determined block of tasks gets assigned to a parallel worker.

Listing 1: Dynamic Load Balancing Method

```
1  my_task      = global_task_counter(
       task_block_size)
2  current_task = 0
3  DO ijkl = 2*ntype, 2, -1
4    DO ij = min(ntype, ijkl-1), max(1, ijkl-ntype)
         , -1
5      kl = ijkl - ij
6      if (my_task .eq. current_task) then
7          call calculate_integral_block()
8          call add_intgrals_to_Fock()
9          my_task = global_task_counter(
               task_block_size)
10     endif
11     current_task = current_task + 1
12   enddo
13 enddo
```

For efficiency, blocks of integrals with similar characteristics are passed to TEXAS integral package. The goal is to enable sharing and reuse of temporary data for integrals in the same block. However, the number of integrals can efficiently be performed concurrently is limited by the cache size and memory requirements. Therefore, we implement two levels of blocking. The higher level is used to assign a block of integrals to the parallel workers. The lower level is used to decide how many integrals can actually be done at the same time in the TEXAS integral package. The integrals are calculated according to the Obara-Saika (OS) method [17]. The algorithm [22], [14] also uses the method proposed by Tracy Hamilton (TRACY) to shift angular momenta from electron 1 (centers 1, 2) to electron 2 (centers 3,4). If the sum of the angular momentum of the quartet shells is less than two, the integrals are handled by a special fast path.

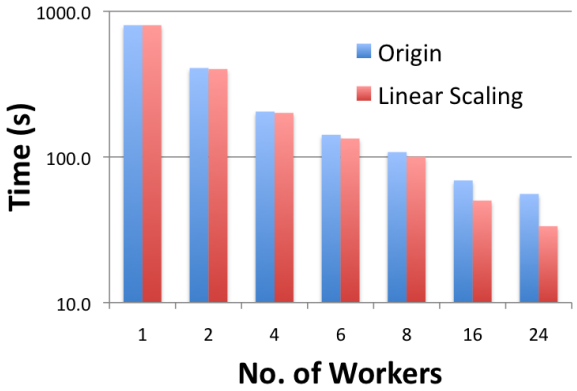

Fig. 2: The initial performance of building Fock matrix and evaluating two-electron integrals on Hopper.

Our performance tuning work focused on the intra-node performance. The inter-node communication of NWChem is handled by a separate software package called the Global Array Toolkit [4]. Optimizing the inter-node communication performance is out of the scope of this paper. Actually, each integral calculation does not involve any communication. Only constructing the Fock matrix does.

IV. Performance Tuning on Hopper

The initial performance of building the Fock matrix and calculating the integrals for the benchmark and the expected perfect linear scaling performance are shown in Figure 2. We note that as the number of workers increases, the running time is reduced linearly. However, for large number of workers scaling performance deteriorates, especially when utilizing all 24 cores/node of the NERSC Hopper system.

A. Task Granularity for Dynamic Load Balancing

Profiling results indicate that the non-perfect linear scaling behavior is related with the dynamic load balancing approach. In order to reduce the number of task requests, we assign a chunk of tasks to each worker (instead of just one). There is a thus a balance between the request overhead and the assigned

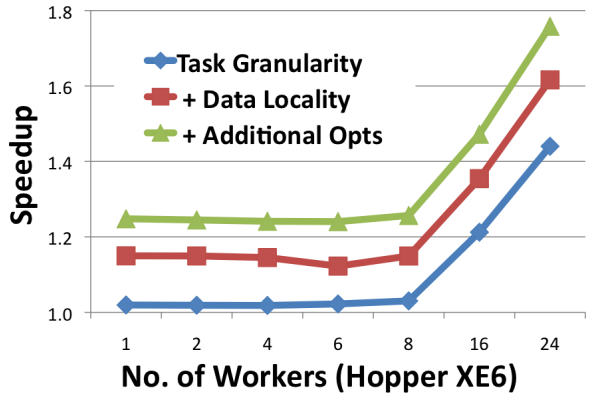

Fig. 3: The cumulative performance improvement relative to the original code on the Cray XE6.

task granularity (smaller granularities result in higher overheads to request those tasks). The baseline version NWChem computes the task granularity using an algorithm based on the number of workers, the request overhead, and the problem characteristics. However, developing a perfect algorithm is challenging due to two reasons. First, the time to finish a task may differ substantially, sometimes even by orders of magnitude. Second, a chunk of tasks assigned to a worker together may further exacerbate the situation.

The granularity size selected by the default algorithm shows inefficiencies when relatively large number of parallel workers are applied. The selected chunk size tends to be too big, leading to the load imbalance. To improve the performance, we modify the default algorithm so that a smaller granularity size can be chosen. Fortunately, the overhead for requesting the tasks increased only slightly. The performance improvement is shown in Figure 3 labeled with Task Granularity. For a fully saturated node of 24 workers, our optimization approach improves performance by over 40%.

B. Data Locality

As mentioned in Section III, the TEXAS integral code does not compute the two-electron integrals one by one. Instead, integrals with similar types are organized in one block and computed together so that intermediate results can be reused. There are some overheads to create the related data structures and perform preliminary computations. If the number of integrals in the block is small, the overhead cannot be effectively amortized. More importantly, array access performance is directly related to this number; small sizes may cause runtimes to suffer due to poor spatial data locality.

By examining the profiling analysis we found that there exist many cases with small block sizes. In extreme cases, the block size is equal to one, leading to poor data locality and low efficiency of the blocking structure. This occurs with the special handling of integrals consisting of only s, p, and sp basis functions. To improve the data locality and the efficiency of the block structure, we updated the code to avoid generating block lengths of one. The performance improvement results are shown in Figure 3 labeled with Data Locality. Results show that performance has been further improved by 10-18% for all configurations. Note that the graph displays cumulative performance improvements.

C. Other Optimizations

We additionally performed several other optimizations. Although the effect of each individual optimization is not significant, the combined effect results in an additional 10% improvement, seen in Figure 3 as "Additional Opts". First, we examined function *uniq_pairs*, which is used to compute the unique ij or kl pairs appeared in a block of integrals and sort them in ascending order. The original implementation first collects all pairs, sorts them using a algorithm similar to quicksort, and stores the unique pairs in an array. However, we found that many pairs are redundant, and therefore developed a simple hash algorithm to partially remove the redundant data first and then sort the remaining data. The number of pairs to be sorted has thus been reduced greatly, resulting in a 75% running time reduction for this function, which accounted for 2-3% of the total program run time.

Other optimizations focused on applying loop transformations to several time-consuming loops. Techniques include loop un-rolling and blocking and data pre-fetching. Although compilers may perform these optimizations automatically, our goal was to help compilers generate more efficient code. The performance impact varied across the loops, sometimes leading to additional small improvements. Finally, we optimized several statements to remove redundant computations.

By combining all these optimization efforts, overall performance improved significantly, by 25-75% relative to the original implementation, as shown in Figure 3.

V. Performance Tuning on Edison

The same set of optimization were also applied to Edison. Figure 4 shows the cumulative performance improvement of our optimization schemes. Similar to Hopper, when more than eight workers are active, the original implementation suffers from load imbalance. By reducing the task granularity, the performance improves up to 50% when 16 workers are used. Optimizing data locality further improved the performance by approximately 13-20%. Finally, other specific optimizations discussed in Section IV resulted in performance gains of another 8%. On Edison, results show that loop transformations, such as unrolling and blocking, has a much smaller performance impact compared with Hopper. This is encouraging for application developers, since these kind of optimizations are not only difficult but also result in less readable code. Overall, the final accumulative performance improvement for all optimizations on Edison resulted in an improvement of 1.2 – 1.8x compared to the original version.

A. Simultaneous Multi-Threading

One major architectural difference between Edison and Hopper is that Edison uses Intel Sandy Bridge processors that supports hyper-threading. Each physical core can thus be

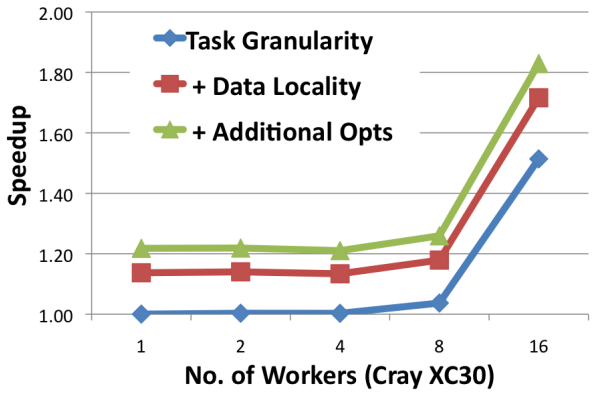

Fig. 4: The cumulative performance improvement relative to the original code on the Cray XC30.

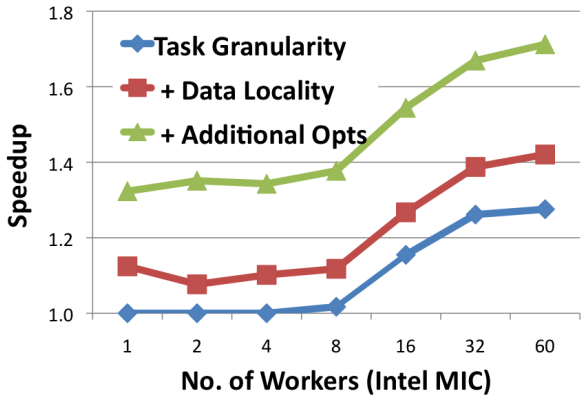

Fig. 5: The cumulative performance improvement relative to the original code on the Intel MIC.

treated as two logical cores, running two threads concurrently, allowing us to run up to 32 workers per node. Using the optimized implementation with hyper-threading, with 32 workers per node, results in an additional performance improvement of approximately 14% compared with 16 workers (in single-threaded mode).

VI. Performance Tuning on Babbage

The Intel MIC architecture supports two programming modes, allowing either execution on the host with selected code sections offloaded to the Intel MIC accelerator (offload mode) or native execution on the Intel MIC architecture (native mode). In this work, we focus our effort to the native mode.

Figure 5 shows the performance improvement on the Intel MIC architecture, which contains 60 physical cores per card. Load balancing is once again a significant bottleneck, and adjusting the task granularity, allows a performance improvement of up to 28% when 60 workers are used. Increasing data locality is additionally effective, further improving the performance about 10% across all shown configurations. Finally, the performance effect of additional optimizations become much more prominent, delivering 20 - 30% performance gains, due mainly to their effects on compiler-driven vectorization. Similar to the Edison platform, manual loop transformations do not show a performance impact.

A. Vectorization

One important feature of the Intel MIC architecture is its vector processing unit. Each core has 32 512-bit-wide vector registers and its vector unit can execute 8-wide double precision SIMD instructions in a single clock, providing 1TFlops/s peak floating-point performance per card. Making effective use of the vector processing unit is the key to obtain high-performance on this architecture, and can be obtained via two approaches. The first is the use of low-level intrinsic functions to explicitly implement vectorization, while the second leverages the compilers and compiler directives to vectorize the code. We focus on the second approach to maintain a general and portable code version (although more extensive vectorization is likely possible via the intrinsic approach). Note that there are some loops that are not vectorizable due to serialization, data dependence, complex code structures, or other reasons. By adding directives or transforming the loop bodies, the compiler is able to vectorize certain key loops. (The effectiveness of the transformations was determined from the vectorization report generated by the Intel compiler's -vec-report6 option.) The transformations include using temporary arrays to assemble data for vectorization, splitting the loop body to separate the vectorizable and non-vectoriable codes.

TABLE II: The total running times for NWChem with and without vectorization on the Intel MIC.

No. of Workers	1	2	4	8	16	32	60
With Vec	4170	2080	1050	517	261	133	72
Without Vec	5380	2610	1310	665	333	167	92

One important parameter that effects the performance of the vector processing units is the vector length. Recall that in the TEXAS implementation, the number of quartet integrals that can be computed together is decided by a block size. This number is carefully decided by TEXAS based on cache size and shell characteristics. It is also the length of a several important data arrays. Increasing this number may potentially improve the vector units performance, while simultaneously degrade data reuse and cache efficiency. We experimented with different sizes and found that the best overall performance is obtained when the default size is quadruped. Further increasing this number causes the cache performance to degrade significantly, especially for the recursive OBASAI functions.

Table II shows impact of vectorization for NWChem by comparing vectorized results with compiler disabled vectorization via the no-vec option. Results show that vectorization improves performance by approximately 22%. For a more detailed analysis, we explored the sequential running times of the top ten subroutines in the TEXAS integral package with and without vectorization in Figure 6, which account for about 73% of the total running times. Function *erintsp* is used to compute the integrals when the sum of the angular

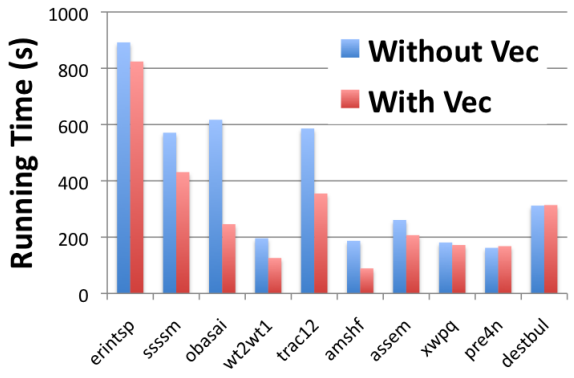

Fig. 6: The running times for the top ten functions in TEXAS integral package with and without vectorization on the Intel MIC.

momentum of the quartet shells is less than two. The other functions, except *destbul*, are used to implement the OBASAI and TRACY algorithms, which are applied when the sum is not less than two. The *destbul* function is used to put the non-zero integral results and corresponding shell labels from a selected block of quartets of shells into a buffer.

- ERINTSP: Most of its time is spent on the FM function which returns the incomplete gamma function fm(x). Based on the value of its input variable x, FM jumps to different branches depending on conditionals. The complex code structure cannot be vectorized directly. By profiling, we discovered that the most often executed branch is the one with time-consuming computation $\frac{1}{\sqrt{(x)}}$. Our optimization thus performed the inverse square root for all the input variables in advance and stored the results in a temporary array. Therefore the results can be directly accessed from the temporary array. The advantage is that the compiler can automatically apply a vectorized implementation of the inverse square root operation. Unfortunately, for some input values, the computation is unnecessary. The final performance results indicate that overall this change is a valuable optimization on the MIC platform.
- SSSSM: Calculates the integrals of the form $(s,s|s,s)(m)$ using the FM function. Compared with `erintsp`, its running time is more heavily dependent on the performance of the inverse square root operations. A similar optimization as in `erintsp` has been performed.
- OBASAI: Computes the integrals of the form $(i+j+k+l,s|s,s)(m)$ or $(s,s|i+j+k+l,s)(m)$ based on the results of `ssssm`. Among the top ten functions, this one benefits most from vectorization, showing an improvement of 2.5x. The innermost loops are dominated by uni-stride data access and can be automatically vectorized by compiler.
- WT2WT1, AMSHF: Are dominated by data movement with segmented uni-stride data access, and require compiler directives to vectorize the code.
- TRAC12: Calculates the integrals of the form $(i+j,s|k+l,s)(m)$ based on `obasai` results by shifting angular momenta from position 1 to 3 or 3 to 1. The innermost loops can be fully vectorized with uni-stride data access. However, some code transformations are required.
- XWPQ, PRE4N, ASSEM: Preprocess or postprocess data for Obasai and Tracy algorithms. Though most loops inside these functions can be vectorized, the data access is dominated by indirect, scattered data access due to integral screening. Memory performance is the bottleneck, not the floating-point operations, thus vectorization has little impact on performance. The improved performance of vectorized `assem` comes from some loops with uni-stride data access patterns.
- DESTBUL: Returns the results of non-zero integrals and corresponding labels. The code can not be easily vectorized due to data dependence.

Of the top ten functions, four routines (`obasai, wt2wt1, trac12, and amshf`) with uni-stride data access benefit directly from vectorization as the compiler can automatically vectorize the code. Some routines require compiler directives to guide the vectorization or perform source level loop transformations. Two other subroutines (`ssssm and erintsp`) need the introduction of temporary arrays to take advantage of the vectorized inverse square root function. Three other subroutines (`assem, xwpq, pre4n`) suffer from indirect, scattered data access and can not easily benefit from vectorization. The final function (`destbul`) cannot be vectorized due to data dependence.

Overall, significant effort has been dedicated to vectorizing each individual subroutine. Our observation is that the process of leveraging vectorization is not only time consuming but also extremely challenging to fully leverage. Future research into new algorithms that are better structured to utilize wide vector units could significantly improve the behavior of the two-electron integral evaluation algorithms.

B. Simultaneous Multi-Threading

The Intel MIC architecture features many in-order cores on a single die, where each core also supports 4-way hyper-threading (4 hardware threads). However, the front-end of the pipeline can only issue up to 2 instructions per cycle, where the two instructions are issued by different threads. Therefore, at least two hardware threads are necessary to achieve the maximum issue rate. By running two workers per physical core (120 threads), using the optimized implementation, the running time can be reduced from 72 to 56 seconds — a 30% improvement. Unfortunately, we can not run more than two workers per physical core due to memory limitation. There is only 8GB memory per card, which seriously limits the total number of workers that can be run concurrently.

VII. Performance Tuning on Mira

On the Mira BG/Q, the Global Array Toolkit [4] is built using communication network ARMCI-MPI. As a result, dy-

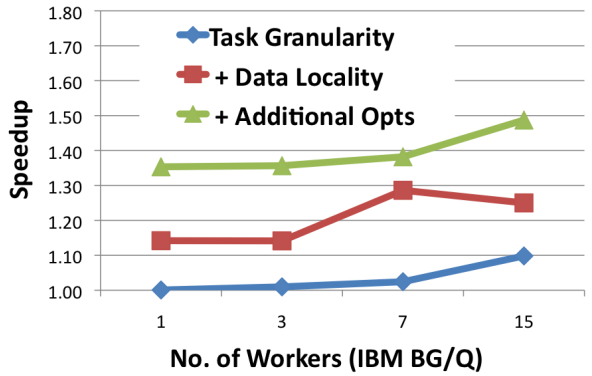

Fig. 7: The cumulative performance improvement relative to the original code on the IBM BG/Q.

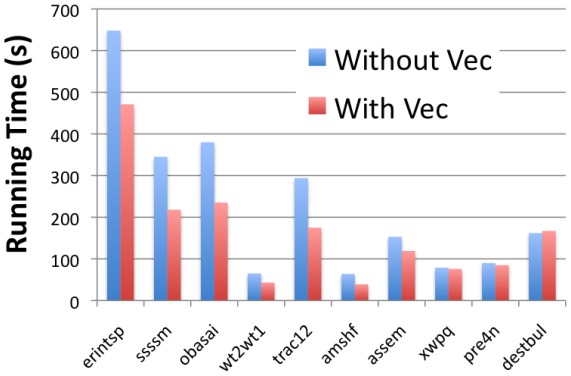

Fig. 8: The running times for top ten functions in TEXAS integral package with and without vectorization on the IBM BG/Q.

namic load balancing becomes very expensive. Initial results showed that more than 30% of the running time is spent on requesting tasks. This is related with MPI-2 RMA operation which does not support atomic operations. To reduce the overhead, we developed a PAMI (Parallel Active Message Interface [13]) interface using PAMI_Rmw function to request tasks. The overhead was therefore reduced to around 10%, however this is still a significant cost compared with other platforms. Using the MU (messaging unit) SPI (system programming interface) may improve performance and will be the subject of future investigation. Instead, we adopt a master-slave approach, which devotes one worker to responding to the task requests and not performing integral calculations. In this way, the overhead of requesting tasks becomes trivial. However, this is offset by the cost of lossing one worker for the integral calculation.

The performance improvement of our optimizations on the 16-core per node platform is shown in Figure 7. Recall that given the dedicated master thread, only an odd number of workers is available. Additionally we successfully applied all previously discusssed optimizations, resulting in an overall performance improvement of 35 - 49%.

A. Vectorization

The BG/Q architecture supports vectorization, where each core has a quad-vector double precision floating point unit called IBM QPX capable of 8 floating point operations per cycle. Compiler directives are provided to inform the compiler the conditions of data dependence and data alignment. However, we observed that the requirements of BG/Q compiler derived vectorization is even more strict than on the Intel MIC architecture. The loops with indirect data access can not be automatically vectorized by the IBM compiler. Nonetheless, comparing the results of vectorized version and non-vectorized version, results show that the vectorized code delivers a 25% improvement compared with the non-vectorized version. Figure 8 shows the running times for top ten subroutines in the TEXAS integral package with and without vectorization on the BG/Q. The pattern of vectorization effect on these subroutines is similar to that on the Intel MIC architecture. The first two subroutines benefit from the vectorization of inverse square root operation, The next four subroutines benefit from vectorization of the uni-stride data access. The next three subroutines suffer from indirect, scattered data access. The last one can not be vectorized due to data dependence. Note that the impact of vectorization also differs depending on the subroutine and architecture (BG/Q versus MIC).

B. Simultaneous MultiThreading

Similar to the Intel MIC architecture, each BG/Q core also supports four hardware threads. Each cycle two instructions can be issued selected from two different threads with one to the XU units and another to the AXU units [1]. XU handles all branch, integer, load/store instructions while AXU executes floating-point instructions. Therefore, at least two threads are needed to maximize the instruction issue rate. Figure 9 displays the NWChem running times when 1, 2, and 4 hardware threads per core are used on a full node. Running 2 workers per core (32 per node) improves performance by a factor of 1.7x, while running 4 workers per core (64 per node) results in a significant improvement of 2.2x — thus demonstrating the effective impact of simultaneous multithreading for NWChem on the BG/Q platform.

VIII. Results & Discussion

The goal of our study is to understand the performance effect of different optimization techniques and find a consistent tuning approach that enables NWChem to perform efficiently on a variety of leading scientific platforms. The architectural diversity of high-end computing systems make this a significant challenge, which was tackled on four important platforms: the Cray XE6, the Cray XC30, the Intel MIC, and the IBM BG/Q

Experimental results show that improving load balance is the most efficient approach to accelerate NWChem performance on all four platforms, followed by increasing data locality. Developing more intelligent algorithms based on

Fig. 9: The performance of running 1, 2, and 4 hardware threads per core on a full IBM BG/Q node.

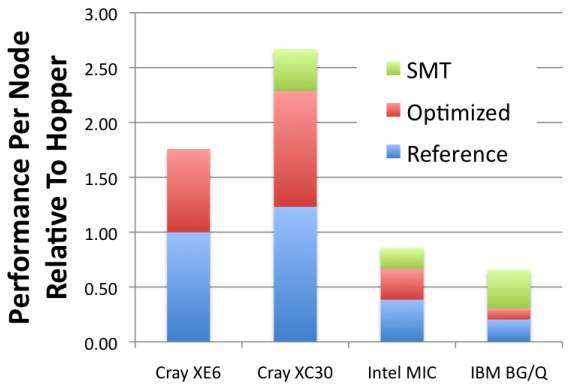

Fig. 10: The node performance relative to Cray XE6.

application insights (such as the hash sorting algorithm) is also effective across all platforms. However, those optimizations and directly related loop transformations, such as loop unrolling and data prefetching, and are only effective on some of the platforms. This is actually a good sign for application developers, as compiler maturity and emerging auto-tuning techniques may relieve users of these burdensome transformations.

Simultaneous multi-threading and vectorization are key architectural features that increasingly are driving high node performance. Our results show that multi-threading is effective in improving NWChem performance. On the IBM BG/Q platform, running 2, and 4 tasks per core can improve performance by 1.7x and 2.2x, respectively. While, on the Intel MIC, running two tasks per core can improve the performance 1.3x. The limited memory size (8GB) prohibited us from running 4 tasks per core (240 threads per node) on MIC. From the design of the instruction issue on the IBM BG/Q and the Intel MIC, two instructions issued in a clock cycle must come from different threads, necessitating multi-threading to achieve high performance.

Vectorization has long been supported by many processor families. However, as recent architectures have moved to higher SIMD lengths, vectorization has become a more critical component of extracting high performance from the underlying hardware. Our experimental results showed that the performance difference between the vectorized code (using compiler directives) and the non-vectorized code is about 22% and 25% MIC and BG/Q respectively. The key to extracting performance from vectorization is to develop uni-stride data access or use vectorized library functions. For NWChem, simply focusing on each individual subroutines and attempting to vectorize the loops is not enough to extract full potential from the vector processing units. A better approach is to develop new algorithmic designs that can maximize the advantages of vectorization.

Finally, Figure 10 presents a summary comparison normalized to the Cray XE6, which highlights original, optimized, and multi-threaded performance (where appropriate). Overall our approach consistently improves the Fock matrix construction and integral calculation for NWChem across all four platforms, attaining up to 2.5x speedup compared with the original version.

IX. Related Work

There are numerous studies examining NWChem optimization schemes. The importance of vectorization for integral calculations had been recognized by Hurley, Huestis, and Goddard [11] in previous work. They described a vectorized procedure to transform integrals over atomic symmetry orbitals into integrals over molecular orbitals on a vector platform Alliant FX8/8. Foster et al. [3] presented scalable algorithms for distributing and constructing the Fock matrix in SCF problems on several massively parallel processing platforms. They also developed a mathematical model for communication cost to compare the efficiency of different algorithms. Tilson et al. [19] compared the performance of TEXAS integral package with the McMurchie-Davidson implementation on IBM SP, Kendall Square KSR-2, Cray T3D, Cray T3E, and Intel Touchstone Delta systems. Ozog et al. [18] explored a set of static and dynamic scheduling algorithms for block-sparse tensor contractions within the NWChem computational chemistry code. Similar algorithms may also be applied for integral calculations. Hammond et al. [6] studied the performance characteristics of NWChem using TAU. However, the focus was on communication. In addition, Jong et al. [12] provided a review of the current state of parallel computational chemistry software utilizing high-performance parallel computing platforms. The major difference of our work is that we focus on performance tuning (as opposed to algorithm development), and developed an effective strategy across four modern HPC platforms.

X. Summary and Conclusions

In this paper, we examined the performance tuning processes for the Fock matrix construction and integral calculations of NWChem on four deployed high performance computing platforms. The results indicate that load balancing has significant potential for improving performance, and attains up to 50% speedup. When combined with our additional

optimization strategies, an overall speedup of up to 2.5x was achieved.

On platform that supports simultaneous multithreading, running multiple threads can improve the performance significantly. On the IBM BG/Q platform, running 2 and 4 threads per core can improve performance by 1.7x and 2.2x, respectively.

Finally, extracting the full performance potential from the vector processing units is a significant challenge for NWChem. Via substantial programming effort, we obtained a vectorized version running approximately 25% faster compared to non-vectorization mode on the MIC and BG/Q platforms. However, the current code can not be fully vectorized due to complex code structures, indirect non-continuous data access, and the true data dependence and serialization. Our future work will focus on developing new algorithms which can more effectively harness the potential of the vector processing units and multithreading on next-generation supercomputing platforms.

XI. Acknowledgements

All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] A2 Processor User's Manual for BlueGene/Q. http://www.alcf.anl.gov/user-guides/ibm-references#a2-processor-manual.

[2] Edison Cray XC30. http://www.nersc.gov/systems/edison-cray-xc30/.

[3] Foster, I., Tilson, J., Wagner, A., Shepard, R., Harrison, R., Kendall, R., and Littlefield, R. Toward High-Performance Computational Chemistry: I. Scalable Fock Matrix Construction Algorithms. *Journal of Computational Chemistry 17* (1996), 109–123.

[4] Global Arrays Toolkit. http://www.emsl.pnl.gov/docs/global/.

[5] Gill, P. M. W. Molecular Integrals Over Gaussian Basis Functions. *Advances in Quantum Chemistry 25* (1994), 141–205.

[6] Hammond, J., Krishnamoorthy, S., Shende, S., Romero, N. A., and Malony, A. Performance Characterization of Global Address Space Applications: A Case Study with NWChem. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE 00* (2010), 1–17.

[7] Harrison, R., Guest, M., Kendall, R., D. Bernholdt, A. Wong, M. S., Anchell, J., Hess, A., Littlefield, R., Fann, G., Nieplocha, J., Thomas, G., Elwood, D., Tilson, J., Shepard, R., Wagner, A., Foster, I., Lusk, E., and Stevens, R. Toward high-performance computational chemistry: II. a scalable self-consistent field program. *Journal of Computational Chemistry 17* (1996), 124–132.

[8] Helgaker, T., Olsen, J., and Jorgensen, P. *Molecular Eletronic-Structure Theory*. Wiley, www.wiley.com, 2013.

[9] Helgaker, T., and Taylor, P. R. *Modern Electronic Structure Theory (Advances in Physical Chemistry)*. World Scientific, www.worldscientific.com, 1995, ch. Gaussian Basis Sets and Molecular Integrals.

[10] Hopper Cray XE6. http://www.nersc.gov/systems/hopper-cray-xe6/.

[11] Hurley, J. N., Huestis, D. L., and Goddard, W. A. Optimized Two-Electron-Integral Transformation Procedures for Vector-Concurrent Computer Architecture. *The Journal of Physical Chemistry 92* (1988), 4880–4883.

[12] Jong, W. A., Bylaska, E., Govind, N., Janssen, C. L., Kowalski, K., Muller, T., Nielsen, I. M., Dam, H. J., Veryazov, V., and Lindh, R. Utilizing High Performance Computing for Chemistry: Parallel Computational Chemistry. *Physical Chemistry Chemical Physics 12* (2010), 6896 – 6920.

[13] Kumar, S., Aamidala, A. R., Faraj, D. A., Smith, B., Blocksome, M., Cernohous, B., Miller, D., Parker, J., Ratterman, J., Heidelberger, P., Chen, D., and Steinmacher-Burrow, B. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *The 26th International Parallel and Distributed Processing Symposium* (May 2012).

[14] Lindh, R., Ryu, U., and Liu, B. The Reduced Multiplication Scheme of the Rys Quadrature and New Recurrence Relations for Auxiliary Function Based Two Electron Integral Evaluation. *The Journal of Chemical Physics 95* (1991), 5889 – 5892.

[15] The Intel MIC. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[16] Mira IBM Bluegene/Q. http://www.alcf.anl.gov/user-guides/mira-cetus-vesta.

[17] Obara, S., and Saika, A. Efficient Recursive Computation of Molecular Integrals Over Cartesian Gaussian Functions. *The Journal of Chemical Physics 84* (1986), 3963 – 3975.

[18] Ozog, D., Shende, S., Malony, A., Hammond, J. R., Dinan, J., and Balaji, P. Inspector-Executor Load Balancing Algorithms for Block-Sparse Tensor Contractions. In *Proceedings of the 27th international ACM conference on International conference on supercomputing* (May 2013).

[19] Tilson, J. L., Minkoff, M., Wagner, A. F., Shepard, R., Sutton, P., Harrison, R. J., Kendall, R. A., and Wong, A. T. High-Performance Computational Chemistry: Hartree-Fock Electronic Structure Calculations on Massively Parallel Processors. *International Journal of High Performance Computing Applications 13* (1999), 291–306.

[20] Top500 Supercomputer Sites. http://www.top500.org/lists/2013/06/.

[21] Valiev, M., Bylaska, E., Govind, N., Kowalski, K., Straatsma, T., van Dam, H., Wang, D., Nieplocha, J., Apra, E., Windus, T., and de Jong, W. Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications 181* (2010), 1477–1489.

[22] Wolinski, K., Hinton, J. F., and Pulay, P. Efficient Implementation of the Gauge-Independent Atomic Orbital Method for NMR Chemical Shift Calculations . *Jounal of the American Chemical Society 112* (1990), 8251 – 8260.