Thread-Level Parallelization and Optimization of NWChem for the Intel MIC Architecture

Hongzhang Shan, Samuel Williams, Wibe de Jong, Leonid Oliker
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA, 94720
hshan, swwilliams, wadejong, loliker@lbl.gov

ABSTRACT

In the multicore era it was possible to exploit the increase in on-chip parallelism by simply running multiple MPI processes per chip. Unfortunately, manycore processors' greatly increased thread- and data-level parallelism coupled with a reduced memory capacity demand an altogether different approach. In this paper we explore augmenting two NWChem modules, triples correction of the CCSD(T) and Fock matrix construction, with OpenMP in order that they might run efficiently on future manycore architectures. As the next NERSC machine will be a self-hosted Intel MIC (Xeon Phi) based supercomputer, we leverage an existing MIC testbed at NERSC to evaluate our experiments. In order to proxy the fact that future MIC machines will not have a host processor, we run all of our experiments in `native` mode. We found that while straightforward application of OpenMP to the deep loop nests associated with the tensor contractions of CCSD(T) was sufficient in attaining high performance, significant effort was required to safely and efficiently thread the TEXAS integral package when constructing the Fock matrix. Ultimately, our new MPI+OpenMP hybrid implementations attain up to 65× better performance for the triples part of the CCSD(T) due in large part to the fact that the limited on-card memory limits the existing MPI implementation to a single process per card. Additionally, we obtain up to 1.6× better performance on Fock matrix constructions when compared with the best MPI implementations running multiple processes per card.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; D.2 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Performance, Concurrent Programming

PMAM'15, February 07-11 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3404-4/15/02 $15.00
http://dx.doi.org/10.1145/2712386.2712391.

Keywords

OpenMP, OpenMP Task, MPI, Thread-level Parallelism, Performance, Manycore Architecture, NWChem, CCSD(T), Fock Matrix Construction, Texas Integral

1. INTRODUCTION

Over the course of the last decade, multicore architectures emerged as the standard for all major HPC computing platforms. Today, manycore and accelerated architecture offer substantially higher performance and may eclipse multicore as the basis for HPC. Unfortunately, both manycore and accelerated architectures demand extreme thread- and data-level parallelism to attain high performance. As existing compilers and runtime systems lack the maturity to automatically extract these forms of parallelism from existing applications, users must express these forms of parallelism explicitly (e.g. OpenMP pragmas and intrinsics) in their codes. To maximize performance, users must consider load balancing, task granularity, software overhead, and many other performance factors. Thus, for many large-scale applications, efficiently exploiting manycore and accelerated architectures can be challenging.

NWChem [24] is a comprehensive open source computational chemistry package for solving challenging chemical and biological problems using large scale ab initio molecular simulations. It has been widely used all over the world to solve a wide range of complex scientific problems. NWChem provides many methods for computing the properties of molecular and periodic systems using standard quantum mechanical descriptions of the electronic wave function or density.

In this work, we focus on two frequently used NWChem modules — CCSD(T) and Fock matrix construction. In the existing implementations of these modules, multicore parallelism is only exploited using MPI; there is no explicit thread-level parallelism. As such, on manycore platforms, the current performance is severely limited as it is often not able to make full use of all the available computing resources due to the constraints of limited memory. Our approach uses OpenMP to express thread-level parallelism for these two modules. In order to proxy the future NERSC-8 supercomputer Cori [4] which is based on a future self-hosted Intel manycore MIC architecture, we use the NERSC testbed Babbage and run in *native* mode. In native mode, one programs the MIC as if it were a self-hosted processor with 60 cores each with 4 hardware thread contexts (240 threads total). Thus, the host is unused, host memory is unavailable, and there are no PCIe transfers. Without a fast, latency-optimized host processor, native-mode configurations can be

very sensitive to Amdhal's Law.

The principle contributions of this work include:

1. Quantifying the impact of limited application parallelism when running on manycore architectures in the context of two different NWChem modules.

2. Evaluating several threading and parallelization techniques that allow one to exploit the full 240 hardware thread contexts on MIC. Previously, CCSD(T) could only exploit a single MPI process per MIC while the Fock matrix construction could only exploit up to 60 MPI processes per MIC.

3. With further optimization, our hybrid MPI+OpenMP codes attain net speedups of 65× and 1.6× relative to the original MPI implementations for the triples part of the CCSD(T) and Fock matrix construction, respectively.

The rest of the paper is organized as follows. After discussing some related work, we describe our evaluation platform and experimental setup. We then proceed to describe the algorithmic details, threading approaches, and other optimizations for the triples part of the CCSD(T) and Fock matrix construction modules in Sections 4 and 5 respectively. Finally, we summarize our results and outline some future work.

2. RELATED WORK

The performance of NWChem has been extensively studied on a variety of compute architectures. However, few of them focus on the thread-level parallelism issue on manycore architectures as we do. In our previous work, we studied how to optimize the performance of the Fock matrix construction on the Intel MIC architecture [21]. That approach was restricted to a flat MPI implementation and focused on improving load balancing and data-level parallelism. A resultant observation was that due to memory constraints, we were not able to run more than 60 MPI processes per MIC card. Although tailoring the code to reduce the memory requirements allowed up to 120 MPI processes, the approach was not a general solution and was not repeated here. Apra et al. studied CCSD(T) performance on the Intel MIC architecture for a large-scale application [1]. In their study, the Intel MIC cards are used in offload mode and not in native mode as we do. As such, their work would not be a good proxy for the NERSC8 Cori supercomputer as one may rely on the fast host processor to avoid performance limitations arising from Amdahl's Law. Nevertheless, we found that similar optimizations could be applied in both native and offload mode. Ma et al. studied CCSD(T) performance on several GPU platforms using hybrid CPU-GPU execution [14, 15]. Ghosh et al. studied the communication performance for TCE [8]. Ozog et al. explored a set of static and dynamic scheduling algorithms for block-sparse tensor contractions within the NWChem computational chemistry code [17].

Liu et al. developed a new scalable parallel algorithm for Fock matrix construction. Their focus was on large heterogeneous clusters [26]. Foster et al. presented scalable algorithms for distributing and constructing the Fock matrix in SCF problems on several massively parallel processing platforms [7]. Tilson et al. compared the performance of TEXAS integral package with the McMurchie-Davidson implementation on the IBM SP, Kendall Square KSR-2, Cray T3D, Cray T3E, and Intel Touchstone Delta systems [23].

With respect to OpenMP performance on the Intel MIC, Carmer et al. evaluated the overhead of OpenMP programming using a couple of simple benchmarks [6], while Schmidl et al. studied the OpenMP performance using kernels and applications and found that porting OpenMP codes to the Intel MIC needs performance tuning [20].

3. EXPERIMENTAL SETUP

Babbage is an Intel MIC testbed at NERSC [2] with 45 compute nodes connected by an Infiniband network. Each node contains two Intel Xeon (host) processors and two MIC (Xeon Phi) cards. Each MIC card contains 60 cores running at 1.05 GHz and 8 GB GDDR memory. Although the theoretical memory bandwidth is 352GB/s, it is nearly impossible to exceed about 170 GB/s using the STREAM benchmark [22]. Each core includes a 32KB L1 cache, a 512KB L2 cache, a 8-way SIMD vector processing unit, supports 4 hardware threads, and provides a peak performance of about 16.8 GFlop/s (1 TFlop/s per chip). Unfortunately, the core can only issue up to two instructions per cycle and then only if there are at least two threads per core. In order to proxy the NERSC8 Cori supercomputer [4], we run in native mode. As such, the Xeon cores cannot be used to hide sequential bottlenecks. Moreover, the PCIe bus is unused and thus does not artificially impair performance. In all experiments, we use the Intel Fortran 64 Compiler XE version 14.0.1 and use *balanced* affinity as it delivered the best performance.

In the paper, we use CCSD(T) and the Fock matrix construction modules as the basis for our evaluation. The input file for CCSD(T) is tce_ccsd2_t_cl2o.nw which can located in the NWChem distributed package. However, we changed the *tile size* from 15 to 24 to improve performance and the basis set from cc-pvdz to aug-cc-pvdz which is augmented with added diffuse functions. For Fock matrix construction, we use the same input file (c20h42.nw) as our previous study [21]. This benchmark is designed to measure the performance of the Hartree-Fock calculations on a single node with a reasonable runtime for tuning purposes.

4. COUPLED CLUSTER TRIPLES ALGORITHM IN CCSD(T)

CCSD(T) is often called the Ògold standardÓ of computational chemistry [19, 18]. It is one method in the Coupled Cluster (CC) family. The coupled cluster methods are widely used in quantum chemistry as a post-Hartree-Fock ab initio quantum chemistry method due to their high accuracy and polynomial time and space complexity [5]. They perform extremely well for the molecular systems as they accurately describe electron correlation part of the interactions.

In this section, we will focus on the triples algorithm in CCSD(T), which is the most computationally expensive component of the calculation. We will discuss the algorithm and then will discuss our approach to OpenMP parallelization and performance optimization.

4.1 Algorithm

The dominant computations in the CCSD(T) algorithm are double-precision tensor contractions. Tensor contrac-

Algorithm 4.1 CCSD(T) Triples Algorithm

```
 1: for p4 = 1 to nvab do
 2:   for p5 = p4 to nvab do
 3:     for p6 = p5 to nvab do
 4:       for h1 = 1 to noab do
 5:         for h2 = h1 to noab do
 6:           for h3 = h2 to noab do
 7:             allocate d_singles and d_doubles (6D tensors)
 8:             call ccsd_t_doubles_l(d_doubles,p4,p5,p6,h1,h2,h3)
 9:             call ccsd_t_singles_l(d_singles,p4,p5,p6,h1,h2,h3)
10:             Sum d_singles & d_doubles into energy1 & energy2
11:           end for
12:         end for
13:       end for
14:     end for
15:   end for
16: end for
```

tions are generalized multidimensional matrix-matrix multiplications. The typical tensor contractions involved in the triples part of the CCSD(T) algorithm can be represented by the following equations that generate a six-dimensional tensor from either the contraction of a two-dimensional and four-dimensional tensor or two four-dimensional tensors shown in Equations 1 and 2. Eventually, the 6D tensors are reduced into a single number. In the triples algorithm, there are 9 tensor contractions that are similar to Equation 1 and 18 like Equation 2.

$$T(p4,p5,p6,h1,h2,h3) = T1(p4,h1) * T2(p5,p6,h3,h2) \quad (1)$$

$$T(p4,p5,p6,h1,h2,h3) = T2(p4,p7,h1,h2) * V2(p5,p6,h3,p7) \quad (2)$$

The multidimensional arrays that represent a tensor (typically 200-2000 for each dimension) are stored in a tiled fashion to enable the distribution of the work over many processors and to limit local memory usage on each processor. The size of the tile will depend on available memory and typically ranges from 10 to 40. We use 24 in our experiments. In practice, tensor operations are often dominated by index permutation and generalized matrix-matrix multiplication.

The pseudocode for the triples algorithm is shown in Algorithm 4.1. Lines 1-6 loop through all the occupied and unoccupied tiles. The loop body contains four major steps (Lines 7-10). Line 7 allocates the temporary buffers for the 6D tensors of Equations 1 and 2. The upper memory requirement for each 6D tensor is $tilesize^6$ doubles (roughly 1.46GB using our tile size of 24). Line 8 fetches the two 4D tensors in Equation 2 from the tile domain space and computes the 6D tensor d_doubles. Similarly, Line 9 fetches the 2D and 4D tensors in Equation 1 and computes the 6D tensor d_singles. Finally, Line 10 sums the two 6D tensors with the appropriate scaling factors and increments the global variables energy1 and energy2.

4.2 Baseline OpenMP Parallelization and Performance

The most computationally intensive part of Algorithm 4.1 is Line 8, where the tensor d_doubles are calculated. This phase therefore often becomes the main target for performance optimization [1, 14, 15]. In particular, Apra et al. [1] have studied the CCSD(T) performance on the Intel MIC architecture, where the MIC node is used as an accelerator, and the OpenMP parallelized d_doubles computation is offloaded to the MIC processor. Their study leveraged the high-performance host processor to compute both d_singles and the sum operation, as well as computing d_doubles. However, for future self-hosted homogeneous manycore processors, running these computations sequentially on a lightweight core can result in a major performance impediment, thus motivating our study.

Algorithm 4.2 Nested Loop to Compute 6D Tensor (d_doubles)

```
 1: !$OMP Parallel do private(p4,p5,p6,p7,h1,h2,h3), collapse(3)
 2: for p5 = 1 to p5d do
 3:   for p6 = 1 to p6d do
 4:     for p4 = 1 to p4d do
 5:       for h1 = 1 to h1d do
 6:         for h3 = 1 to p3d do
 7:           for h2 = 1 to p2d do
 8:             for p7 = 1 to p7d do
 9:               triplesx(h2,h3,h1,p4,p6,p5) +=
10:               t2sub(p7,p4,h1,h2) * v2sub(p7,h3,p6,p5)
11:             end for
12:           end for
13:         end for
14:       end for
15:     end for
16:   end for
17: end for
```

In practice, the high memory requirements of CCSD(T) often precludes running more than one process per MIC processor. This motivates entirely threading CCSD(T) to ensure efficient execution on homogeneous manycore systems. In order to highlight this importance, we evaluated performance by threading either d_doubles, or all three major steps (d_doubles, d_singles, as well as summation). Figure 1 presents the overall performance as a function of the number of OpenMP threads. There is one thread per core from 1-60 threads, two at 120, three at 180, and four at 240 threads. Figure 1 conclusively shows that failing to thread d_singles and the summation results in nearly a 2.1× loss in performance. As such, it is obvious that it is imperative one thread all of these routines when running on a homogenous manycore processor. We discuss our approach and optimization here.

The computation of the 6D tensors on Lines 8-9 of Algorithm 4.1 includes two major substeps: calculating the lower dimensional tensors with correct order and computing the 6D tensors. Computing the 6D tensors is implemented using a deep loop nest. Based on type of contraction, there are 9 different loop structures in ccsd_t_singles_l and 18 in ccsd_t_doubles_l (27 total). Algorithm 4.2 shows an example contraction from ccsd_t_doubles_l, where *h1*, *h2*, and *h3* are the occupied spin-orbital indices and *p4*, *p5*, *p6*, and *p7* are the unoccupied spin-orbital indices. The two 4D tensors *t2sub* and *v2sub* are sub-blocks of cluster amplitude and two-electron tensors, respectively. The 6D tensor *triples* (another name for d_doubles) is the projections of the tiles.

The most straightforward approach to threading Algorithm 4.2 is to simply add an `!$OMP parallel do` directive to the outermost loop. We use the `collapse` clause to increase the total number of loop iterations threaded at a

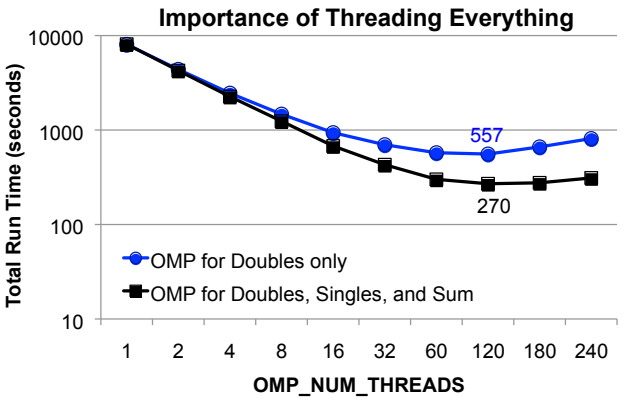

Figure 1: Benefit of progressive OpenMP threading in CCSD(T) on MIC in Native mode. Whereas an accelerated system can rely on a fast host processor, it is essential that one thread Doubles, Singles, *and* the Summation when using a homogeneous many-core processor.

Algorithm 4.3 Original 6D Tensor Sum Implementation

```
 1: i = 0
 2: for p4 = 1 to range_p4 do
 3:   d4 = array(offset_p4 + p4)
 4:   for p5 = 1 to range_p5 do
 5:     d5 = array(offset_p5 + p5)
 6:     for p6 = 1 to range_p6 do
 7:       d6 = array(offset_p6 + p6)
 8:       for h1 = 1 to range_h1 do
 9:         d1 = array(offset_h1 + h1)
10:         for h2 = 1 to range_h2 do
11:           d2 = array(offset_h2 + h2)
12:           for h3 = 1 to range_h3 do
13:             d3 = array(offset_h3 + h3)
14:             d = 1.0 / ((d1+d2+d3) - (d4+d5+d6))
15:             energy1 += factor*d*d_doubles(i)*d_doubles(i)
16:             energy2 += factor*d*d_doubles(i)*
                  (d_doubles(i)+d_singles(i))
17:             i = i + 1
18:           end for
19:         end for
20:       end for
21:     end for
22:   end for
23: end for
```

time — an essential step as the small value of $p5d$ would otherwise lead to underutilization of the 240 threads on MIC. This directive was applied to all 27 nested loops.

The summation of Line 10 of Algorithm 4.1 is shown in Algorithm 4.3. Unfortunately, the variable i impedes the compiler from correctly threading this loop nest. As such, we rewrote the loop nest to be thread-safe (Algorithm 4.4). In addition to the OpenMP parallelization, we optimize the performance further by extracting the common variable "factor" in Lines 15-16 of Algorithm 4.3 out of the loops and multiply it only once after the whole computation as in Lines 28-29 of Algorithm 4.4.

Algorithm 4.4 OpenMP Parallelized 6D Tensor Sum (Line 10)

```
 1: e1= e2 = 0.0
 2: !$OMP Parallel do private(p4,p5,p6,h,h2,h3), col-
    lapse(3)
 3: private(d4,d5,d6,d1,d2,d3,d, e1, e2, offset,nom, i)
        reduction(+:e1, e2)
 4: for p4 = 1 to range_p4 do
 5:   for p5 = 1 to range_p5 do
 6:     for p6 = 1 to range_p6 do
 7:       d4 = array(offset_p4 + p4)
 8:       d5 = array(offset_p5 + p5)
 9:       d6 = array(offset_p6 + p6)
10:       offset = p6-1+range_p6*(p5-1+range_p5*(p4-1))
11:       for h1 = 1 to range_h1 do
12:         for h2 = 1 to range_h2 do
13:           for h3 = 1 to range_h3 do
14:             d1 = array(offset_h1 + h1)
15:             d2 = array(offset_h2 + h2)
16:             d3 = array(offset_h3 + h3)
17:             d = 1.0 / ((d1+d2+d3) - (d4+d5+d6))
18:             i = h3-1+range_h3*(h2-1+
                  range_h2*(h1-1+range_h1*offset))
19:             e1 = e1+d*d_doubles(i)*d_doubles(i)
20:             e2 = e2+d*d_doubles(i)*(d_doubles(i)+d_singles(i))
21:           end for
22:         end for
23:       end for
24:     end for
25:   end for
26: end for
27: !$OMP end parallel do
28: energy1 = energy1 + e1 * factor
29: energy2 = energy2 + e2 * factor
```

Figure 2 shows more details of the scalability of the initial (unoptimized) OpenMP performance. There is one thread per core from 1-60 threads, two at 120, three at 180, and four at 240 threads. The reduction in total run time scales almost linearly up to 16 OpenMP threads. The line labeled *Loop Nests* represents the total time spent across the 27 deep loop nests plus the summation loop. These operations scale well to 120 threads beyond which the benefits of HyperThreading are asymptotic. Conversely, the time spent fetching the lower dimensional tensors (*GetBlock*) shows no improvement as it was not initially threaded. Unfortunately, it appears that HyperThreading elsewhere has a deleterious effect on this routine. As such, the baseline implementation attains its best performance with 120 threads. The best times for each line have been labeled.

4.3 Performance and Further Optimization

To further improve the code performance, we implemented the following optimizations listed below. The resultant optimized nested loop implementation for Algorithm 4.2 is shown in Algorithm 4.5.

1. Parallelize tce_sort: To reduce memory consumption, the 2D and 4D tensors are divided into tiles and stored in a complex hash space [3, 8]. Once fetched, their indices need to be permuted to proper order by calling function tce_sort. We apply the OpenMP *parallel do*

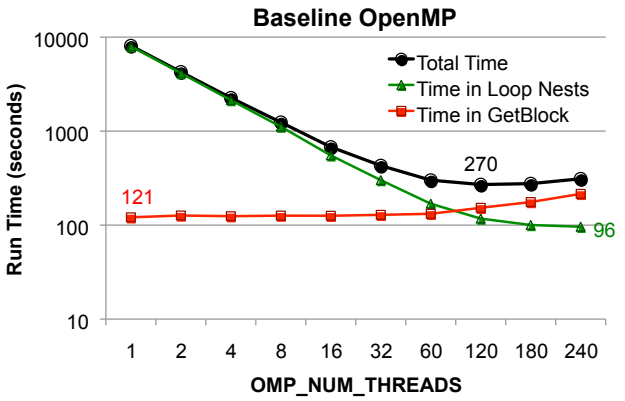

Figure 2: Baseline OpenMP CCSD(T) run time. There is one thread per core through 60 cores. For 120-240 threads, HyperThreading is exploited. Observe the effects of the *GetBlock* sequential bottleneck. Labels record time in seconds.

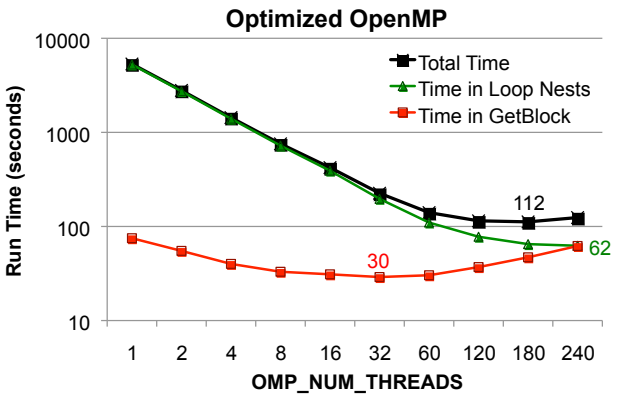

Figure 3: Optimized OpenMP CCSD(T) run time. The performance for the nested loops has been significantly improved. The *GetBlock* sequential bottleneck is mitigated so long as there is only one thread per core. Overall, we achieve a 2.5× speedup. Labels record time in seconds.

directive to parallelize this sorting process. This sorting time is included in the *GetBlock* time.

2. Optimize get_block_ind: Using optimized nested loops to replace generalized sorting function tce_sortacc. For some cases, we simplify the loop body statement from : $sorted(i) = sorted(i) + unsorted(j)$ to $sorted(i) = unsorted(j)$ to avoid reading array *sorted*.

3. Reorder the indices for 2D and 4D tensors: As shown in Algorithm 4.2, the index order of the tensor *t2sub* is *p7*, *p4*, *h1*, *h2*, while the nested loop is ordered as *p7*, *h2*, *h1*, *p4* from inner to outsider. The different index and loop order will cause noncontiguous data access, resulted in lower memory performance. To improve the data locality, we permute the index for tensor *t2sub* so that the index order becomes the same as they appear in the nested loops.

Algorithm 4.5 Optimized Nested Loops to Compute 6D Tensor (d_doubles)

```
1: !$OMP Parallel do private(p6p5,p7,h1p4,h2,h3), collapse(2)
2: for p6p5 = 1 to p6d * p5d do
3:   for h1p4 = 1 to h1d * p4d do
4:     for h3 = 1 to p3d do
5:       for h2 = 1 to p2d do
6:         for p7 = 1 to p7d do
7:           !DIR$ ASSUME_ALIGNED triplesx: 64
8:           !DIR$ ASSUME_ALIGNED t2sub: 64
9:           !DIR$ ASSUME_ALIGNED v2sub: 64
10:          !DIR$ LOOP COUNT AVG=24
11:          triplesx(h2,h3,h1p4,p6p5) +=
12:                    t2sub(p7,h2,h1p4)*v2sub(p7,h3,p6p5)
13:        end for
14:      end for
15:    end for
16:  end for
17: end for
```

4. Merge adjacent loop indices to increase the number of iterations.

5. Align the array data to 64 bytes.

6. Exploit OpenMP loop control directives.

Figure 3 presents the resultant benefit of our optimizations. Compared with the baseline OpenMP implementation of Figure 2, the *Loop Nests* performance has been improved about 1.54× while the best *GetBlock* time has been improved from 121 seconds to 30 seconds — with an overall performance gain of 2.5×. The best total running time is obtained when 180 threads are used. Compared to the original flat MPI implementation that could only run one process per card due to memory constraints, our approach resulted in a 65× improvement in run time. Unfortunately, further performance gains are impeded by the *GetBlock* bottleneck that is related with Global Array [10]; optimization of the Global Array implementation is beyond the scope of this paper.

5. FOCK MATRIX CONSTRUCTION

The Fock matrix construction is the core computational operation of the widely used Hartree-Fock (HF) method [11, 12, 9] which is a fundamental approach for solving the Schrödinger equation in quantum computing and is often used as the starting point for the accurate but more time consuming electronic correlation methods such as the coupled cluster approach we described in Section 4. Efficient parallelization of the Fock matrix construction can be much more challenging than optimizing CCSD(T). To that end, we will begin by discussing the algorithm and its challenges and then explore different approaches to exploiting thread-level parallelism in combination with load balancing techniques on MIC.

5.1 Algorithm

In the HF algorithm, the Fock matrix (F) must be repeatedly constructed. The Fock matrix is a square $N \times N$

matrix where N is the number of the basis functions used to describe the system. Each element ij is updated using the following equation [7]:

$$F_{ij} = h_{ij} + \sum_{k=1}^{N}\sum_{l=1}^{N} D_{kl}((ij|kl) - \frac{1}{2}(ik|jl)) \quad (3)$$

where h is one-electron Hamiltonian, D is the one-particle density matrix, and $(ij|kl)$ is a six dimensional integral, called a two-electron repulsion integral (ERI). As each element of the $N \times N$ Fock matrix requires one calculate $O(N^2)$ two-electron integrals, the naive time required for these integrals is computationally prohibitive. However, by applying some screening for small values as well as exploiting molecular and permutation symmetry, the total complexity can be reduced to between $O(N^2)$ and $O(N^3)$.

Algorithm 5.1 Fock Matrix Construction — Original Implementation

```
 1: current_task_id = 0
 2: my_task = global_task_counter(task_block_size)
 3: for ij = 1 to nij do
 4:   for kl = 1 to nij do
 5:     if (my_task .eq. current_task_id) then
 6:       Prep_quartet_list_for_integral_calc(ij,kl,qlist,pinfo,plist,...)
 7:       Calculate_integrals_using_TEXAS(qlist,results,scratch)
 8:       Update_Fock_matrix_using_integral_results(fock,results)
 9:       my_task=global_task_counter(task_block_size)
10:     end if
11:     current_task_id = current_task_id + 1
12:   end for
13: end for
```

The dominating cost in the construction of the Fock matrix is the calculation of the two-electron integrals. The primary integral driver for NWChems [24] is the TEXAS integral package. The 70-thousand line TEXAS integral package [25] computes quadruple integrals in blocks (chunks). Although computing the large number of integrals makes Fock matrix construction numerically very expensive, each computation is independent. Unfortunately, as any screening and symmetry are intertwined with the integral calculations, the actual time to compute an ERI may differ several orders of magnitude. Worse, varying angular momentums of the corresponding basis functions can further exacerbate the variability. To cope with this execution variability, NWChem uses a shared global task counter (essentially an efficient task queue) to dynamically load balance work among MPI processes. In order to minimize network pressure on the global task counter, tasks are doled out in blocks.

Algorithm 5.1 presents the pseudocode for the Fock matrix construction. The variable *nij* is the number of blocks of pairs of shells ij. Lines 3-4 loop through all pairs of blocks. Line 6 constructs the quartet list using the pairs from blocks ij and kl based on the pair information stored in array *pinfo, plist* and other data structures. Once the quartet list *qlist* has been created, the TEXAS integral package in Line 7 is invoked to perform the integrals using the Obara-Saika (OS) method [16, 25, 13] with the results stored into the array *results*. For efficiency, integrals with similar characteristics are computed together in order to maximize sharing and reuse of temporary data. As each integral may affect a number of Fock matrix elements related by values of i, j, k, and l, the update on Line 8 is a potential impediment to straightforward threading.

5.2 Thread safety of TEXAS integral routines

NWChem and the TEXAS integral package is 15-year old legacy fortran code that makes extensive use of common blocks to pass variables. In order to ensure this code is thread-safe, one must declare OpenMP attributes (e.g. *shared* or *threadprivate*) for every variable in every common block. If a common block appears in multiple subroutines, the variable attributes should be defined for every occurrence. In some cases, this necessitated partitioning a common block. For example, the common block *pnl002* includes four variables. Among them, *ncshell*, *ncfunct*, *nblock2* should be defined as shared (by default) and *integ_n0* should be separated from the original common block and defined as threadprivate. The corresponding codes have been shown below.

```
Original:
common /pnl002/ ncshell,ncfunct,nblock2,integ_n0

OpenMP:
common /pnl002/ ncshell,ncfunct,nblock2
common /pnl0022/ integ_n0
c$OMP threadprivate(/pnl0022/)
```

5.3 OpenMP Parallelization and Optimization

We explored three different OpenMP parallelization approaches to exploiting thread-level parallelism on MIC. These approaches span a spectrum that trades programability for performance through massive thread-level parallelism. The first approach OpenMP directives are added to the computationally expensive TEXAS integral routines to allow for fine grain parallelization. The second approach parallelizes the code in a coarse-grained manner so that each OpenMP thread will essentially perform the same work as an additional MPI process would have. Unfortunately, significant programming effort is required to make the code thread safe. Finally, we use the OpenMP task model to overcome this overhead and inefficiency.

5.3.1 Approach #1: OpenMP Parallelization of the TEXAS integral routines

For Approach #1 OpenMP directives are used to enable threading of the loop structures within the individual routines of the TEXAS integral package. In our previous study, we identified the top ten subroutines in the TEXAS integral package that accounted for about 75% of the Fock matrix construction time [21]. Although these routines use various loop nests that can be threaded, they tend to be short, much more complex and irregular in code structures and data access patterns. The first attempt at threading these routines involved placing *c$OMP parallel do* or *c$OMP do* directives on these loops where appropriate. As the *c$OMP parallel do* directive has slightly more overhead than *c$OMP do* directive, we aggregate parallel regions and use multiple *c$OMP do* directives. Every variable inside the threaded loops must define their OpenMP attribute correctly (*shared*, *private*, *first private*, etc.). Loop collapsing is difficult to be applied here because of the data dependence between the loop indices.

5.3.2 Approach #2: OpenMP Parallelization at the Fock Matrix Construction Level

Algorithm 5.2 Fock Matrix Construction — OpenMP Implementation

```
 1: c$OMP parallel private(mytid, current_task_id,
    my_task, ij, kl, qlist, results, scratch, ...)
 2: c$OMP shared(pinto, plist, ...)
 3: c$OMP reduction (+: fock)
 4: current_task_id = 0
 5: c$OMP critical
 6: my_task = global_task_counter(task_block_size)
 7: c$OMP end critical
 8: for ij = 1 to nij do
 9:   for kl = 1 to nij do
10:     if (my_task .eq. current_task_id) then
11:       Prep_quartet_list_for_integral_calc(ij,kl,qlist,pinfo,plist,...)
12:       Calculate_integrals_using_TEXAS(qlist,results,scratch)
13:       Update_Fock_matrix_using_integral_results(fock,results)
14:       c$OMP critical
15:       my_task=global_task_counter(task_block_size)
16:       c$OMP end critical
17:     end if
18:     current_task_id = current_task_id + 1
19:   end for
20: end for
21: c$OMP end parallel
```

Approach #2 attempts to replicate the task parallelism of each of the MPI ranks of the Fock matrix construction routine using threads. Although structurally similar, this approach has the advantage that it should use significantly less memory than simply adding more MPI processes on a chip. Moreover, unlike Approach #1, the coarse-grained parallelism employed by this approach (one thread per task) minimizes any OpenMP overheads for the TEXAS integral package. To accomplish this approach, one must address three challenges — ensuring thread-safety of common blocks (see 5.2), extending the existing process-based dynamic load balancing to OpenMP threads, and ensuring the Fock matrix can be efficiently updated. The resultant OpenMP implementation of the new dynamic load balancing algorithm is shown in Algorithm 5.2.

Approach #2 still depends on the use of the `global_task_counter` to provide dynamic load balancing among OpenMP threads. As this call necessitates MPI communication by multiple threads, the function must be called from within an OpenMP Critical section and MPI must be initialized at the level of `MPI_THREAD_SERIALIZED`.

With multiple threads independently calculating ERI's, there is the possibility that two threads will simultaneously attempt to update the same Fock matrix element. In the MPI implementation, this data hazard is avoided by creating independent copies of the Fock matrix. Although using an OpenMP critical section or atomic updates could address this data hazard, the performance penalties are severe. Although OpenMP locks (e.g. one per row of the Fock matrix) seemed to be an attractive solution, the overhead coupled with the sheer number of updates per ERI resulted in impaired performance. Ultimately, the best solution was to mimic the MPI implementation at the OpenMP level. That is, each thread receives a copy of the Fock matrix. These copies are reduced to the master using a `reduction(+:fock)` clause.

Algorithm 5.3 Fock Matrix Construction — OpenMP Task Implementation

```
 1: Allocation of myfock for each thread
 2: c$OMP parallel
 3: myfock() = 0
 4: c$OMP master
 5: current_task_id = 0
 6: my_task = global_task_counter(task_block_size)
 7: for ij = 1 to nij do
 8:   for kl = 1 to nij do
 9:     if (my_task .eq. current_task_id) then
10:       c$OMP task firstprivate(ij,kl) default(shared)
11:       create_task(ij,kl,myfock, ...)
12:       c$OMP end task
13:       my_task=global_task_counter(task_block_size)
14:     end if
15:     current_task_id = current_task_id + 1
16:   end for
17: end for
18: c$OMP end master
19: c$OMP taskwait
20: c$OMP end parallel
21: Explicit Reduction on myfock to Fock matrix
22:
23: subroutine create_task(ij,kl,myfock, ...)
24:   Prep_quartet_list_for_integral_calc(ij,kl,qlist,pinfo,plist,...)
25:   Calculate_integrals_using_TEXAS(qlist,results,scratch)
26:   Update_Fock_matrix_using_integral_results(myfock,results)
27: end subroutine
```

5.3.3 *Approach #3: Using OpenMP Task Directives in Fock Matrix Construction*

In Approach #3 OpenMP in the Fock matrix construction is exploited using the OpenMP task model to dynamically assign work to threads. Algorithm 5.3 illustrates our implementation. The code begins by creating an OpenMP parallel region with per-thread copies of the Fock matrix (*myfock*). The per-thread copies are explicitly allocated in advance and initialized to 0, instead of implicility as done in Algorithm #2. The master thread then traverses the loop iteration space spawning OpenMP tasks that calculate ERIs and update their own copy of the Fock matrix. Observe that only one thread will call `global_task_counter` thereby minimizing contention. After all tasks have been completed, a explicit hand-coded reduction is performed to fold the per-thread copies of the Fock matrix into the master copy. NWChem manages memory itself using a preallocate stack and we leverage this functionality, guarded with an OpenMP atomic directive, to allocate the thread-private variables like *qlist* and *scratch*. Restoration of this stack is easily facilitated upon completion of all tasks.

5.4 Performance Comparison and Tuning

Figure 4 presents the total execution time for each of the ten most important subroutines of the TEXAS integral package as well as the total running times (total) under five different parallelization strategies — flat MPI with 60 processes, and using the hybrid implementation of Approach #1 (fine-grained threading of the TEXAS integrals) with 60 processes using 1, 2, 3, or 4 threads per process (HyperThreading on a core). Note, although hybrid with 60 processes and 1 thread per process expresses no more par-

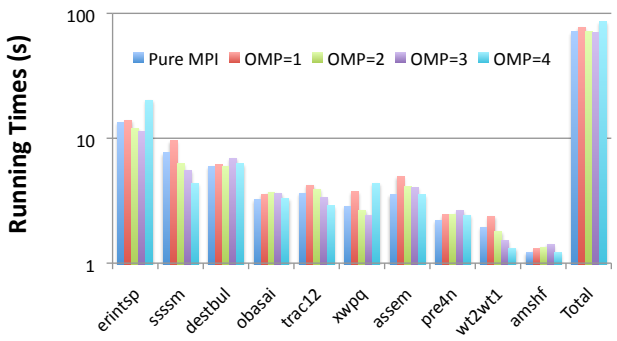

Figure 4: The total time spent in each of the top ten subroutines in the TEXAS integral package (and the total running time) using Approach #1 (loop-level threading) as a function of the number of OpenMP threads per process. In all cases, there are 60 MPI processes. Note, Pure MPI is not the same as MPI with one thread per process as the latter incurs wasted overhead for each `omp parallel` region.

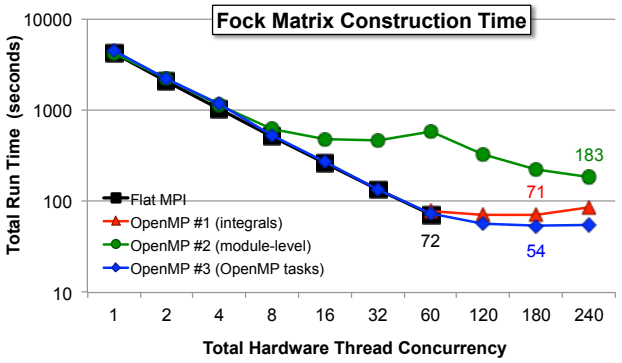

Figure 5: The performance and scalability of the hybrid MPI+OpenMP Approaches #1, #2, #3 compared with original flat MPI implementation. The flat MPI implementation is limited to 60 processes, while the OpenMP implementations can use all 240 hardware thread contexts. While approach #1 uses 60 MPI processes with 1,2,3, or 4 OpenMP threads per MPI process, both Approach #2 and #3 use 1 MPI process with up to 240 OpenMP threads.

allelism than flat MPI, it does incur additional overhead for each OpenMP parallel region. The 60 process limit is an artifact of the high memory requirements per process and the limited memory per MIC card. As one can see, although the benefit of threading varies significantly from loop to loop, the net benefit using this style of OpenMP parallelization is minimal.

There are several reasons why the fine-grained approach to OpenMP parallelization provided less of a benefit than it did for CCSD(T). First, each OpenMP parallel region requires some overhead. Thus, with one OpenMP thread per process, there is only additional overhead with no parallelization benefit. Second, the total time spent in these routines is much less than the time spent in the similar CCSD(T) operations. As such, even with multiple threads per process, it is difficult to amortize the initial overhead. Third, the depth of the iterations for each loop is relatively small. When combined with the fact that data dependencies across loops prevent collapsing the nested loops, we find it is difficult to efficiently thread and vectorize the routines. Finally, the data access pattern is much less regular than the loop nests in CCSD(T). This prevents easy vectorization and may incur much more cache coherency transactions.

Figure 5 compares the scalability of all three OpenMP approaches to the flat MPI implementation. Approach #1 (threading in the integrals) with one OpenMP thread tracks the performance of the flat MPI implementation perfectly up to 60 processes. Beyond 60 MPI processes, Approach #1 uses 2, 3, or 4 threads per process to provide threading within the TEXAS integrals. As discussed at the beginning of this section, the lack of performance gain in threading the integrals (Figure 4) leads to no performance gain of Approach #1 beyond 60 MPI processes — hardly an ideal use of a manycore processor.

In contrast to the flat MPI with 60 procceses, Approach #2 and #3 use 1 MPI process with 1 to 240 threads. All implementations scale well to 8 cores. Unfortunately, at that point, Approach #2 (parallelization at the module-level) saturates. However, beyond 60 threads, it exploits HyperThreadding and can see some benefit as it fully exploits each MIC core. This strange performance is due to several factors. Although, the OpenMP critical section that safe guards task dissemination causes some serialization overhead but not significant. Nevertheless, most of the additional time is spent in the preparation stage and is likely due to inefficiencies in the OpenMP run time's management and reduction of the long list of potentially large private variables.

Unlike Approach #2, Approach #3 (OpenMP Tasks) continues to scale from 8 through 60 cores and tracks the flat MPI performance perfectly. Beyond 60 cores, exploiting HyperThreading through the OpenMP task model allows NWChem to make full use of the MIC processor. Unlike Approach #2, the ERI preparation time has been greatly reduced and is no longer an impediment. Ultimately, with 1 process of 180 threads, the OpenMP task implementation outperforms the flat MPI implementation by 1.33×.

Thus far, when using the OpenMP task model, we have always fixed the number of MPI processes at 1 and simply varied the number of OpenMP threads. In order to find the globally optimal balance between threads and processes, we benchmarked all combinations of MPI ranks and threads per core. Figure 6 presents the resultant performance. In all cases, we have fixed the total concurrency to 60, 120, 180, or 240 hardware thread contexts (i.e. 60 cores with 1, 2, 3, or 4 threads per core). We observe that the best performance can be obtained using all 240 thread contexts configured as 4 processes each of 60 threads. Moreover, this process of tuning the balance of threads and processes provided a 22% improvement over the fully threaded (1 process of 240 threads) configuration and 1.64× faster than the original flat MPI implementation.

6. SUMMARY AND FUTURE WORK

Unlike multicore architectures which one could exploit by

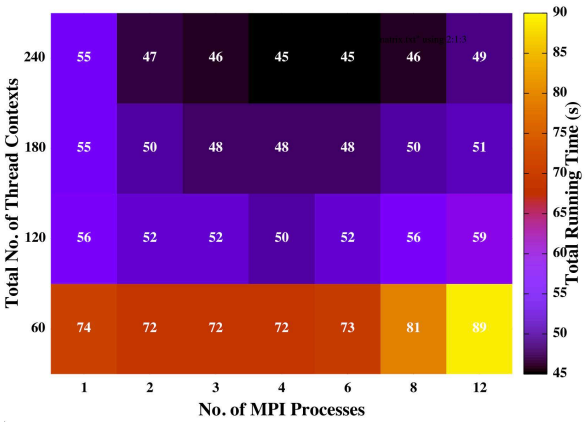

Figure 6: Performance of the OpenMP task implementation as a function of the number of threads and processes. Note, in all cases, we have fixed the total concurrency to 60, 120, 180, or 240 hardware thread contexts (i.e. 60 cores with 1, 2, 3, or 4 threads per core). The best performance is obtained with 4 MPI processes of 60 threads and is 1.64× faster than the original flat MPI implementation.

simply running multiple processes per chip, manycore architectures require a concerted effort to restructure legacy codes to exploit the massive degree of thread-level parallelism. In this paper, we investigated how to restructure two NWChem modules, the triples part of the CCSD(T) and Fock matrix construction, using portable OpenMP directives in order to exploit the full capability of the Intel MIC architecture. In order to proxy the NERSC8 homogenous manycore supercomputer, we perform all calculations in native mode. Unlike the offload model in which MIC is treated like an accelerator and one can exploit the fast host processor for sequential computations, in native mode, all computations must be efficiently threaded in order to avoid any sequential bottlenecks.

Threading the the triples part of the CCSD(T) is relatively straightforward. One can apply the OpenMP directives directly to each tensor contraction. Nevertheless, naive threading was insufficient and some optimization was necessary. The result is far superior to the existing flat MPI implementation (which was constrained to a single process) as it delivers 65× the performance. Conversely, attempting a similar approach for the calculation of the TEXAS two-electron integrals used to construct the Fock matrix provides no benefit. Applying thread-parallelism in a coarse-grained approach provided a more efficient use of hardware. However, the specifics on how to realize coarse-grained parallelism with an inherent lack of thread safety within the module, the presence of high temporary data requirements, and data hazards that impede threading are subtle. Ultimately, we found that use of the OpenMP task model provided a succinct, portable, high-performance solution that allowed the Fock matrix construction routines to exploit the full hardware capability of the MIC processor and delivered a 1.6× speedup over the existing flat MPI implementation. Future work will continue to the process of threading other NWChem packages with OpenMP. Unfortunately, this process of restructuring the code to be thread-friendly was particularly time-consuming and error-prone. Looking forward, tools to remedy this portability gap for legacy codes are essential.

Acknowledgements

Hongzhang Shan, Sam Williams, and Lenny Oliker were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. Wibe de Jong was supported by Intel as part of its Parallel Computing Centers effort and by Laboratory Directed Research and Development (LDRD) funding from Berkeley Lab. 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. This work is supported by Intel as part of its Parallel Computing Centers effort.

7. REFERENCES

[1] E. Apra, M. Klemm, and K. Kowalski. Efficient Implementation of Many-body Quantum Chemical Methods on the Intel Xeon Phi Coprocessor. *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[2] `https://www.nersc.gov/users/computational-systems/testbeds/babbage/`.

[3] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93:276–292, February 2005.

[4] `https://www.nersc.gov/users/computational-systems/nersc-8-system-cori/`.

[5] `http://en.wikipedia.org/wiki/Coupled_cluster/`.

[6] T. Cramer, D. Schmidl, M. Klemm, and D. A. Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. *In Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, November 2012.

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

[8] P. Ghosh, J. F. Hammond, S. Ghosh, and B. Chapman. Performance analysis of the NWChem TCE for different communication patterns. *4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13)*, 2013.

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

[10] http://www.emsl.pnl.gov/docs/global/.

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

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

[13] R. Lindh, U. Ryu, and B. Liu. 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*, 95:5889 – 5892, 1991.

[14] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski. GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *Journal of Chemical Theory and Computation 7(5):1316-1328. doi:10.1021/ct1007247*.

[15] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal. Optimizing Tensor Contraction Expressions for Hybrid CPU-GPU Execution. *Cluster Computing (2013) 16(1):131-155. doi:10.1007/s10586-011-0179-2*, 2013.

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

[17] D. Ozog, S. Shende, A. Malony, J. R. Hammond, J. Dinan, and P. Balaji. 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.

[18] G. D. purvis III and R. J. Bartlett. A full coupled cluster singles and doubles model: The inclusion of disconnected triples. *J. Chem. Phys.*, 76:1910–1918, February 1982.

[19] J. Rezac, L. Simova, and P. Hobza. CCSD[T] Describes Noncovalent Interactions Better than the CCSD(T), CCSD(TQ), and CCSDT Methods. *J. Chem. Theory Comput.*, 9:364–369, 2013.

[20] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Muller. Assessing the performance of OpenMP programs on the intel xeon phi. *In Proceeding Euro-Par'13 Proceedings of the 19th international conference on Parallel Processing*, 2013.

[21] H. Shan, B. Austin, W. A. de Jong, L. Oliker, N. Wright, and E. Apra. Performance Tuning of Fock Matrix and Two-Electron Integral Calculations for NWChem on Leading HPC Platforms. *4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13)*, 2013.

[22] http://www.cs.virginia.edu/stream/ref.html.

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

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

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

[26] X.Liu, A. Patel, and E. Chow. A New Scalable Parallel Algorithm for Fock Matrix Construction. *IEEE International Parallel & Distributed Processing Symposium 2014 (IPDPS14)*, 2014.