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

October 2014

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.

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 [25] 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 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 Intel manycore MIC architecture [16], 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.

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 [22]. 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. 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 [9]. Ozog et al. explored a set of static and dynamic scheduling algorithms for block-sparse tensor contractions within the NWChem computational chemistry code [18].

Liu et al. developed a new scalable parallel algorithm for Fock matrix construction. Their focus was on large heterogeneous clusters [27]. 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 [24].

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 [21].

3 Experimental Setup

Babbage is an Intel MIC testbed at NERSC [2,16] 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 [23]. 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 and PCIe are unused and do not affect 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 [22]. 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 gold standard of computational chemistry [19,20]. 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.

Algorithm 4.1 CCSD(T) Triples Algorithm

```
1:  for $p4 = 1$ to $nvab$ do                                  ▷ nvab: number of unoccupied titles
2:      for $p5 = p4$ to $nvab$ do
3:          for $p6 = p5$ to $nvab$ do
4:              for $h1 = 1$ to $noab$ do                      ▷ noab: number of occupied titles
5:                  for $h2 = h1$ to $noab$ do
6:                      for $h3 = h2$ to $noab$ do
7:                          A: allocate space for d_singles, d_doubles (6D tensors)
8:                          B: call ccsd_t_doubles_l(d_doubles,p4,p5,p6,h1,h2,h3)
9:                          C: call ccsd_t_singles_l(d_singles,p4,p5,p6,h1,h2,h3)
10:                         D: Sum d_singles and d_doubles into energy1 and energy2
11:                     end for
12:                 end for
13:             end for
14:         end for
15:     end for
16: end for
```

4.1 Algorithm

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

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

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

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 labeled A-D. Step A 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). Step B fetches the two 4D tensors in Equation 2 from the tile domain space and computes the 6D tensor d_doubles. Similarly, Step C fetches the 2D and 4D tensors in Equation 1 and computes the 6D tensor d_singles. Finally, Step D sums the two 6D tensors with the appropriate scaling factors and increments the global variables energy1 and energy2.

Algorithm 4.2 Nested Loop to Compute 6D Tensor (exemplar from Step B)

```
!$OMP Parallel do private(p4,p5,p6,p7,h1,h2,h3), collapse(3)
for $p5 = 1$ to $p5d$ do
    for $p6 = 1$ to $p6d$ do
        for $p4 = 1$ to $p4d$ do
            for $h1 = 1$ to $h1d$ do
                for $h3 = 1$ to $p3d$ do
                    for $h2 = 1$ to $p2d$ do
                        for $p7 = 1$ to $p7d$ do
                            triplesx(h2,h3,h1,p4,p6,p5) +=
                                    t2sub(p7,p4,h1,h2)*v2sub(p7,h3,p6,p5)
                        end for
                    end for
                end for
            end for
        end for
    end for
end for
```

4.2 Baseline OpenMP Parallelization and Performance

The high memory requirements of this method often preclude running more than one process per MIC processor. We are thus highly motivated to thread the code to maximize performance. As the tensor contractions can dominate the run time, we focus our initial effort there.

The computation of the 6D tensors in steps B and C of Algorithm 4.1 includes two major substeps: getting the lower dimensional tensors with proper 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 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 step D of Algorithm 4.1 is shown in Algorithm 4.3. Unfortunately, the variable i impeded the compiler from correctly threading this loop nest. As such, we rewrote the loop nest to be

Algorithm 4.3 Original 6D Tensor Sum Implementation (Step D)

```
i = 0
for $p4 = 1$ to $range\_p4$ do
    d4 = array(offset_p4 + p4)
    for $p5 = 1$ to $range\_p5$ do
        d5 = array(offset_p5 + p5)
        for $p6 = 1$ to $range\_p6$ do
            d6 = array(offset_p6 + p6)
            for $h1 = 1$ to $range\_h1$ do
                d1 = array(offset_h1 + h1)
                for $h2 = 1$ to $range\_h2$ do
                    d2 = array(offset_h2 + h2)
                    for $h3 = 1$ to $range\_h3$ do
                        d3 = array(offset_h3 + h3)
                        d = 1.0 / ((d1+d2+d3) - (d4+d5+d6))
                        energy1 += factor*d*d_doubles(i)*d_doubles(i)
                        energy2 += factor*d*d_doubles(i)*(d_doubles(i)+d_singles(i))
                        i = i + 1
                    end for
                end for
            end for
        end for
    end for
end for
```

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 Implementation (Step 6D)

```
 1: e1= e2 = 0.0
 2: !$OMP Parallel do private(p4,p5,p6,h,h2,h3), collapse(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 1 displays 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.

4.3 Performance and Further Optimization

To further improve the code performance, we implemented the following optimizations listed below. Please note, optimizations 3-6 have been previously implemented by researchers in [1]. However,

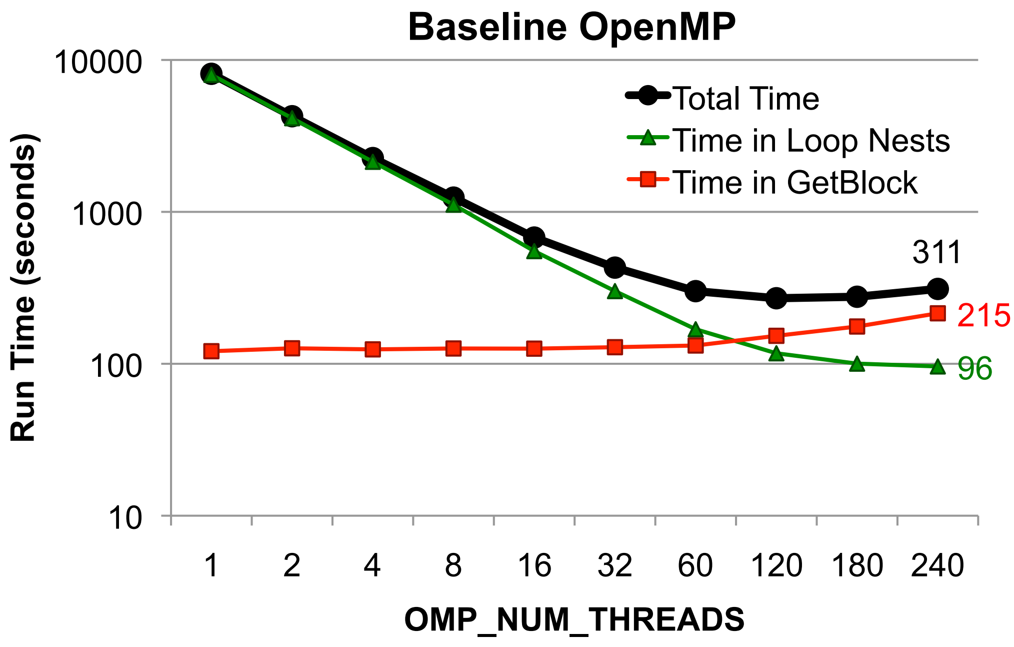

Figure 1: Baseline OpenMP CCSD(T) run time. There is one thread per core through 60 cores. For 120-240 threads, HyperThreading is exploited. Observe the effects of the *GetBlock* sequential bottleneck. Labels record time in seconds.

there are a couple of differences. First, in their study, the Intel MIC nodes are used in offload mode as computing accelerators while in our study they are programmed in native mode. Second, they only apply the optimizations to the tensor contractions in ccsd_t_doubles_l while we apply them to the tensor contractions in ccsd_t_singles_l as well. Finally, we also parallelize the *sum* section. 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, 9]. Once fetched, their indices need to be permuted to proper order by calling function tce_sort. We apply the OpenMP *parallel do* 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.

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.

Algorithm 4.5 Optimized Nested Loops to Compute 6D Tensor (exemplar from Step B)

```
!$OMP Parallel do private(p6p5,p7,h1p4,h2,h3), collapse(2)
for $p6p5 = 1$ to $p6d * p5d$ do
    for $h1p4 = 1$ to $h1d * p4d$ do
        for $h3 = 1$ to $p3d$ do
            for $h2 = 1$ to $p2d$ do
                for $p7 = 1$ to $p7d$ do
                    !DIR$ ASSUME_ALIGNED triplesx: 64
                    !DIR$ ASSUME_ALIGNED t2sub: 64
                    !DIR$ ASSUME_ALIGNED v2sub: 64
                    !DIR$ LOOP COUNT AVG=24
                    triplesx(h2,h3,h1p4,p6p5) +=
                            t2sub(p7,h2,h1p4)*v2sub(p7,h3,p6p5)
                end for
            end for
        end for
    end for
end for
```

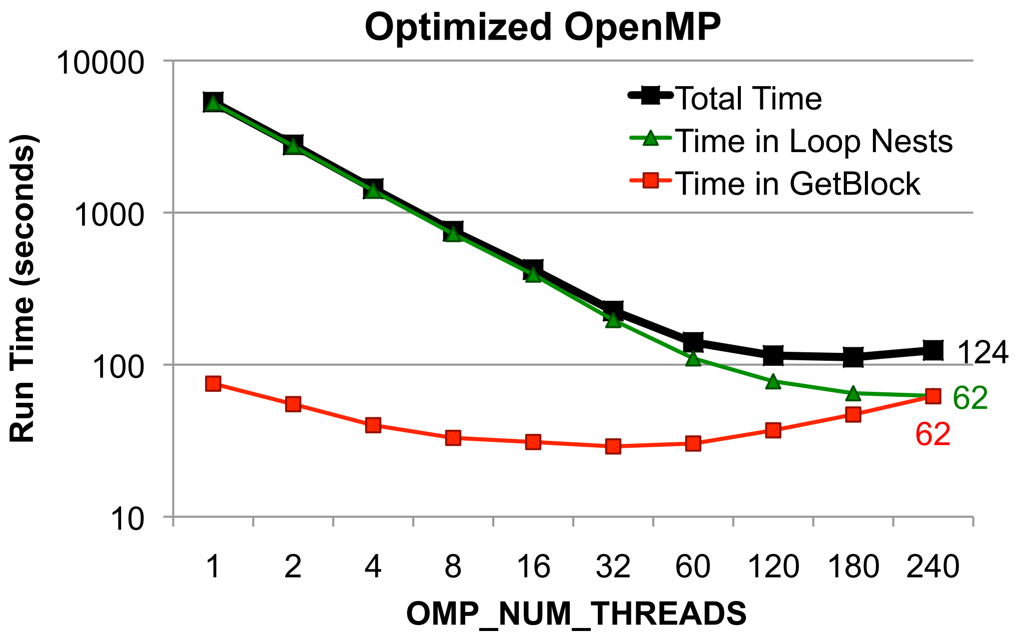

Figure 2: Optimized OpenMP CCSD(T) run time. The performance for the nested loops has been significantly improved. The *GetBlock* sequential bottleneck is mitigated so long as there is only one thread per core. Overall, we achieve a 2.5× speedup. Labels record time in seconds.

Figure 2 presents the resultant benefit of our optimizations. Compared with the baseline OpenMP implementation of Figure 1, the *Loop Nests* performance has been improved about 1.54× while the *GetBlock* time has been improved up to 4.5×. The overall performance has been improved by 2.5×. Compared to the original flat MPI implementation that due to limited memory could only run one process per card, we have improved performance by 65×. Unfortunately, further performance gains are impeded by the *GetBlock* bottleneck which is related with Global Array [8]. And optimization of the Global Array implementation is beyond the scope of this paper.

5 Fock Matrix Construction

Our second benchmark for evaluating performance on MIC is the Fock matrix construction and the associated TEXAS integral package. The Fock matrix construction is the core computational operation of the widely used Hartree-Fock (HF) method [10–12] which is a fundamental approach for solving the Schrödinger's 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 three different approaches to exploiting thread-level parallelism 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]:

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

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 $ijkl = 2 * ntype$ to 2 step $-1$ do
4:     for $ij = min(ntype, ijkl - 1)$ to $max(1, ijkl - ntype)$ step $-1$ do
5:         $kl = ijkl - ij$
6:         if (my_task .eq. current_task_id) then
7:             prepare_quartet_list_for_integral_calculations(ij,kl,qlist, pinfo, plist, ...)
8:             calculate_integrals_using_TEXAS_package(qlist,results,scratch)
9:             update_Fock_matrix_using_integral_results(fock,results)
10:            my_task=global_task_counter(task_block_size)
11:        end if
12:        current_task_id = current_task_id + 1
13:    end for
14: end for
```

In NWChem [25], the Fock matrix construction uses the TEXAS integral package for the two-electron integrals. The 70-thousand line TEXAS integral package [26] 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 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 *ntype* is the number of blocks of pairs of shells ij. Lines 3-5 loop through all pairs of blocks in an order that guarantees that shells with higher angular momentum will be visited first. Line 7 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 is invoked to perform the integrals using the Obara-Saika (OS) method [13, 17, 26] 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 9 is a potential impediment to straightforward threading.

5.2 OpenMP Parallelization and Optimization

As the optimal approach to OpenMP parallelization depends heavily on algorithm and architecture, we explored three different 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 directly parallelizes the computational loops with OpenMP directives. 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.

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

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 [22]. Although these routines are deep loop nests reminiscent of those in CCSD(T), they are 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.

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

In contrast to Approach #1 in which OpenMP is used to thread the loop nests within the individual routines of the TEXAS integral package, Approach #2 attempts to replicate the task parallelism 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 affect this approach, one must address three challenges — ensuring the TEXAS integral package

Algorithm 5.2 Fock Matrix Construction — OpenMP Implementation

```
c$OMP parallel private(mytid, current_task_id, my_task, ijkl, ij, kl, qlist, results, scratch, ...)
c$OMP shared(pinto, plist, ...)
c$OMP reduction (+: fock)
current_task_id = 0
c$OMP critical
mytid = omp_get_thread_num()
c$OMP end critical
my_task = global_task_counter(task_block_size)
for $ijkl = 2 * ntype$ to 2 step $-1$ do
    for $ij = min(ntype, ijkl - 1)$ to $max(1, ijkl - ntype)$ step $-1$ do
        $kl = ijkl - ij$
        if (my_task .eq. current_task_id) then
            prepare_quartet_list_for_integral_calculations(ij,kl,qlist, pinfo, plist, ...)
            calculate_integrals_using_TEXAS_package(qlist,results,scratch)
            update_Fock_matrix_using_integral_results(fock,results)
            c$OMP critical
            my_task=global_task_counter(task_block_size)
            c$OMP end critical
        end if
        current_task_id = current_task_id + 1
    end for
end for
c$OMP end parallel
```

is thread-safe, 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.

The TEXAS integral package is a 20-year old legacy fortran code which 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 blocks. 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 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/)
```

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, MPI must be initialized at at least the `MPI_THREAD_SERIALIZED` level and the function must be called from within an OpenMP Critical section.

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.

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

The third approach to exploiting OpenMP in the Fock matrix construction is to use 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*), which is allocated in advance and initialized to 0. The master thread then traverses the loop iteration space spawning OpenMP tasks that calculate ERIs and update the 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 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.

Algorithm 5.3 Fock Matrix Construction — OpenMP Task Implementation

```
c$OMP parallel
myfock() = 0
c$OMP master
current_task_id = 0
mytid = omp_get_thread_num()
my_task = global_task_counter(task_block_size)
for $ijkl = 2 * ntype$ to 2 step $-1$ do
    for $ij = min(ntype, ijkl - 1)$ to $max(1, ijkl - ntype)$ step $-1$ do
        $kl = ijkl - ij$
        if (my_task .eq. current_task_id) then
            c$OMP task firstprivate(ij,kl) default(shared)
            create_task(ij,kl, ...)
            c$OMP end task
            my_task=global_task_counter(task_block_size)
        end if
        current_task_id = current_task_id + 1
    end for
end for
c$OMP end master
c$OMP taskwait
c$OMP end parallel
Perform Reduction on myfock to Fock matrix

subroutine create_task(ij,kl, ...)
    prepare_quartet_list_for_integral_calculations(ij,kl,qlist, pinfo, plist, ...)
    calculate_integrals_using_TEXAS_package(qlist,results,scratch)
    update_Fock_matrix_using_integral_results(myfock,results)
end subroutine
```

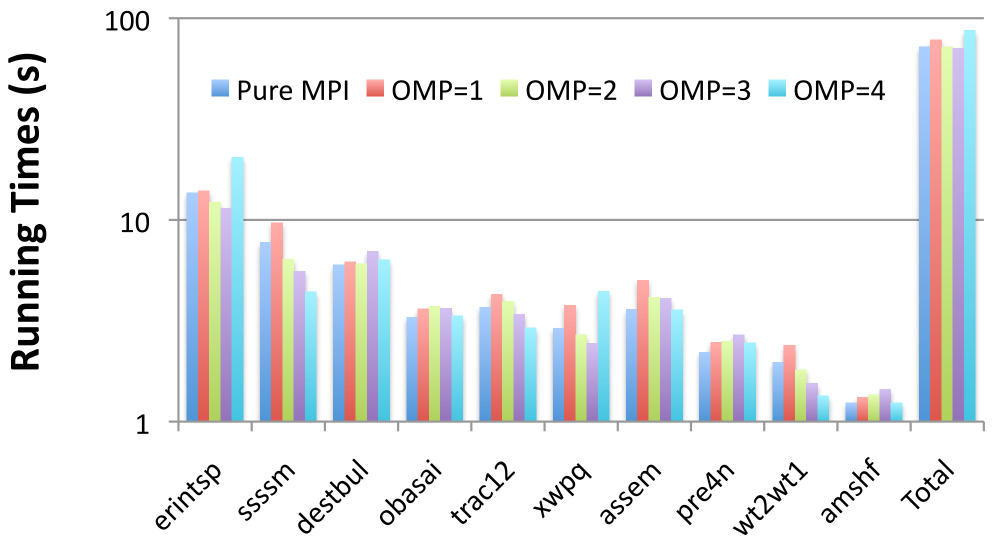

Figure 3: The total time spent in each of the top ten subroutines in the TEXAS integral package (and the total running time) using Approach #1 (loop-level threading) as a function of the number of OpenMP threads per process. In all cases, there are 60 MPI processes. Note, Pure MPI is not the same as MPI with one thread per process as the latter incurs wasted overhead for each `omp parallel` region.

5.3 Performance Comparison and Tuning

Using Approach #1 (threading the loops), Fig. 3 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 the hybrid implementation 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 parallelism 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 number of 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.

Fig. 4 compares the scalability of all three OpenMP approaches to the flat MPI implementation. Note, the flat MPI implementation and Approach #1 both use 60 processes while the latter uses 1, 2, 3, or 4 threads per process. Approach #2 and #3 use 1 process of between 1 and 240 threads. All

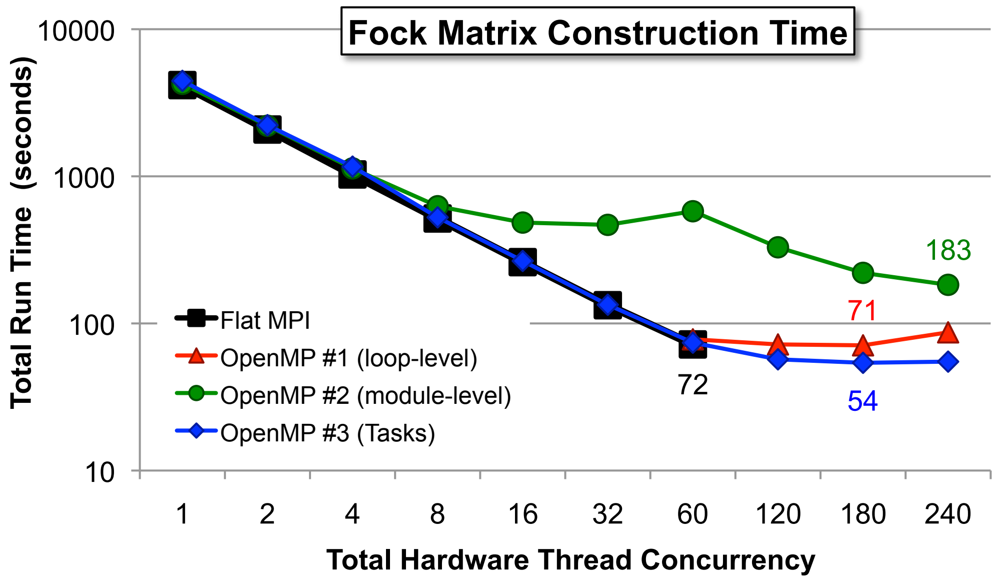

Figure 4: The performance and scalability of the hybrid MPI+OpenMP Approaches #1, #2, #3 compared with original flat MPI implementation. The flat MPI implementation is limited to 60 processes, while the OpenMP implementations can use all 240 hardware thread contexts. While approach #1 uses 60 MPI processes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

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. Moreover, the total time spent in the TEXAS package to compute the quadruple integrals is similar to the flat MPI cases. 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.

As noted, Approach #1 (threading the loop nests) uses flat MPI through 60 cores and then switches to OpenMP. As such, its performance tracks the flat MPI implementation perfectly to that point. By using three threads per process, it delivers a little over 1% better performance than flat MPI — hardly ideal use of a manycore processor.

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 benchmark all combinations. Fig. 5 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,

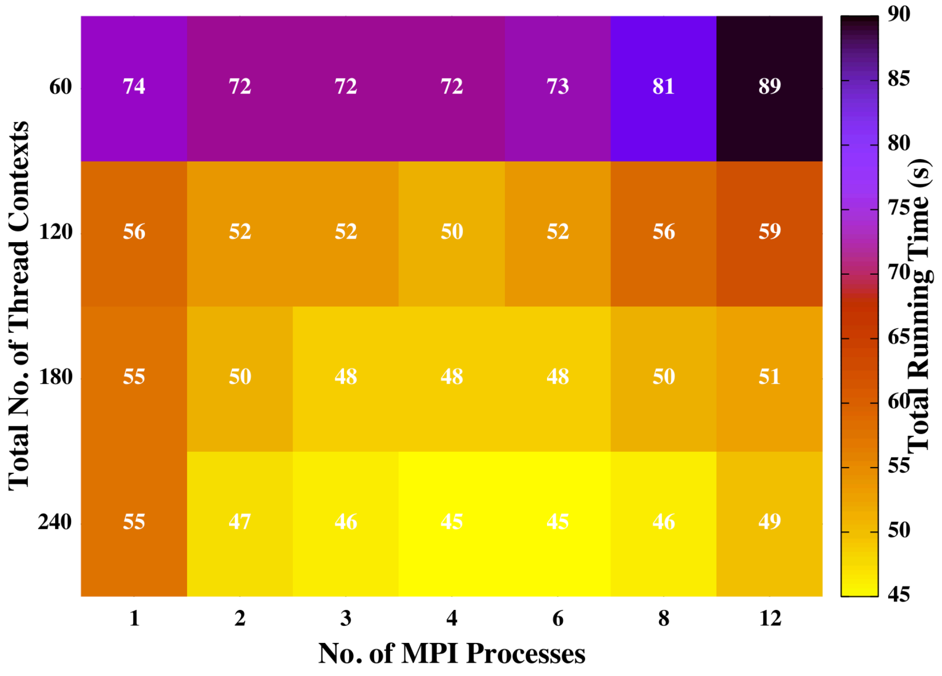

Figure 5: Performance of the OpenMP task implementation as a function of the number of threads and processes. Note, in all cases, we have fixed the total concurrency to 60, 120, 180, or 240 hardware thread contexts (i.e. 60 cores with 1, 2, 3, or 4 threads per core). The best performance is obtained with 4 MPI processes of 60 threads and is 1.64× faster than the original flat MPI implementation.

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 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. For the triples part of the CCSD(T), this process is relatively straightforward. One can apply the OpenMP directives directly to each tensor contraction. This approach 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 electron repulsive integrals used to construct the Fock matrix provides no benefit. Ultimately, 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. 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.

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] Babbage Testbed. `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] Cori Cray XC30. `https://www.nersc.gov/users/computational-systems/nersc-8-system-cori//`.

[5] Coupled cluster method. `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] Global Arrays Toolkit. `http://www.emsl.pnl.gov/docs/global/`.

[9] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] H. Shan, B. Austin, W. D. 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.

[23] STREAM benchmark. `http://www.cs.virginia.edu/stream/ref.html`.

[24] 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.

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

[26] 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.

[27] 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.