

# Scaling up Hartree–Fock Calculations on Tianhe-2

Edmond Chow,<sup>†\*</sup> Xing Liu,<sup>†</sup> Sanchit Misra,<sup>‡</sup> Marat Dukhan,<sup>†</sup> Mikhail Smelyanskiy,<sup>‡</sup>  
Jeff R. Hammond,<sup>‡</sup> Yunfei Du,<sup>§</sup> Xiang-Ke Liao,<sup>§</sup> Pradeep Dubey<sup>‡</sup>

<sup>†</sup>School of Computational Science and Engineering, Georgia Institute of Technology, USA

<sup>‡</sup>Parallel Computing Lab, Intel Corporation, USA

<sup>§</sup>National University of Defense Technology, China

## Abstract

This paper presents a new optimized and scalable code for Hartree–Fock self-consistent field iterations. Goals of the code design include scalability to large numbers of nodes, and the capability to simultaneously use CPUs and Intel Xeon Phi coprocessors. Issues we encountered as we optimized and scaled up the code on Tianhe-2 are described and addressed. A major issue is load balance, which is made challenging due to integral screening. We describe a general framework for finding a well balanced static partitioning of the load in the presence of screening. Work stealing is used to polish the load balance. Performance results are shown on Stampede and Tianhe-2 supercomputers. Scalability is demonstrated on large simulations involving 2938 atoms and 27394 basis functions, utilizing 8100 nodes of Tianhe-2.

## 1 Introduction

Given a set of atomic positions for a molecule, the Hartree–Fock (HF) method is the most fundamental method in quantum chemistry for approximately solving the electronic Schrödinger equation [1]. The solution of the equation, called the wavefunction, can be used to determine properties of the molecule. The solution is represented indirectly in terms of a set of  $n$  basis functions, with more basis functions per atom leading to more accurate, but more expensive approximate solutions. The HF numerical procedure is to solve an  $n \times n$  generalized nonlinear eigenvalue problem via self-consistent field (SCF) iterations.

Practically all quantum chemistry codes implement HF. Those that implement HF with distributed computation include NWChem [2], GAMESS [3], ACESIII [4] and MPQC [5]. NWChem is believed to have the best parallel scalability [6, 7, 8]. The HF portion of this code, however, was developed many years ago with the goal of scaling to one atom per core, and HF in general has never been shown to scale well beyond hundreds of nodes. Given current parallel machines with very large numbers of cores, much better scaling is essential. Lack of scalability inhibits the ability to simulate large molecules in a reasonable amount of time, or to use more cores to reduce time-to-solution for small molecules.

The scalability challenge of HF can be seen in Figure 1, which shows timings for one iteration of SCF for a protein-ligand test problem. Each iteration is composed of two main computational components: constructing a Fock matrix,  $F$ , and calculating a density matrix,  $D$ . In the figure, the dotted lines show the performance of NWChem for these two components. Fock matrix construction stops scaling after between 64 and 144 nodes. The density matrix calculation, which involves an eigendecomposition, and which is often believed to limit scalability, also does not scale but only requires a small portion of the time relative

---

\*Email: echow@cc.gatech.edu

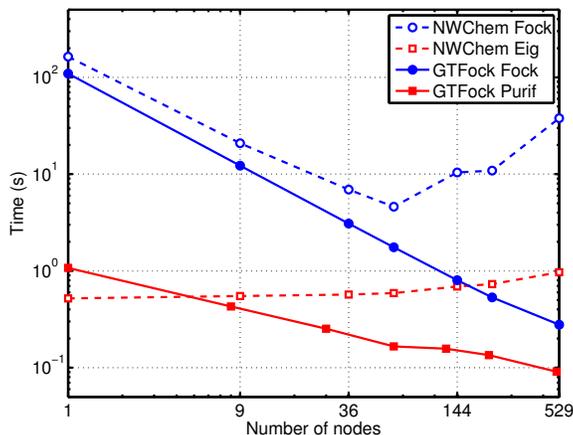


Figure 1: Comparison of NWChem to GTFock for 1hsg\_28, a protein-ligand system with 122 atoms and 1159 basis functions. Timings obtained on the Stampede supercomputer (CPU cores only).

to Fock matrix construction. Its execution time never exceeds that of Fock matrix construction due to the poor scalability of the latter. The figure also shows a preview of the results of the code, GTFock, presented in this paper. Here, both Fock matrix construction and density matrix calculation (using purification rather than eigendecomposition) scale better than in NWChem.

An outline of the SCF algorithm, using traditional eigendecomposition to calculate  $D$ , is shown in Algorithm 1. For simplicity, we focus on the restricted Hartree–Fock version for closed shell molecules. The most computationally intense portions just mentioned are the Fock matrix construction (line 6) and the diagonalization (line 9). In the algorithm,  $C_{occ}$  is the matrix formed by  $n_{occ}$  lowest energy eigenvectors of  $F$ , where  $n_{occ}$  is the number of occupied orbitals. The overlap matrix,  $S$ , the basis transformation,  $X$ , and the core Hamiltonian  $H_{core}$ , do not change from one iteration to the next and are usually precomputed and stored. The algorithm is terminated when the change in electronic energy  $E$  is less than a given threshold.

---

**Algorithm 1:** SCF algorithm

---

- 1 Guess  $D$
  - 2 Compute  $H^{core}$
  - 3 Diagonalize  $S = U_s U_s^T$
  - 4 Form  $X = U_s^{1/2}$
  - 5 **repeat**
  - 6     Construct  $F$ , which is a function of  $D$
  - 7     Form  $F' = X^T F X$
  - 8     Compute energy  $E = \sum_{k,l} D_{kl} (H_{kl}^{core} + F_{kl})$
  - 9     Diagonalize  $F' = C' \epsilon C'^T$
  - 10     $C = X C'$
  - 11    Form  $D = C_{occ} C_{occ}^T$
  - 12 **until** converged
- 

In this paper, we present the design of a scalable code for HF calculations. The code attempts to better exploit features of today’s hardware, in particular, heterogeneous nodes and wide-SIMD CPUs and accelerators. The main contributions of this paper are in four areas, each described in its own section.

- Scalable Fock matrix construction. In previous work, we presented an algorithm for load balancing

and reducing the communication in Fock matrix construction and showed good performance up to 324 nodes (3888 cores) on the Lonestar supercomputer [9]. However, in scaling up this code on the Tianhe-2 supercomputer, scalability bottlenecks were found. We describe the improvements to the algorithms and implementation that were necessary for scaling Fock matrix construction on Tianhe-2. We also improved the scalability of nonblocking Global Arrays operations used by GTFock on Tianhe-2. (Section 3.)

- Optimization of integral calculations. The main computation in Fock matrix construction is the calculation of a very large number of electron repulsion integrals (ERIs). To reduce time-to-solution, we have optimized this calculation for CPUs and Intel Xeon Phi coprocessors, paying particular attention to wide SIMD. (Section 4.)
- Heterogeneous task scheduling. For heterogeneous HF computation, we developed an efficient heterogeneous scheduler that dynamically assigns tasks simultaneously to CPUs and coprocessors. (Section 5.)
- Density matrix calculation. To improve scalability of density matrix calculation, we employ a purification algorithm that scales better than diagonalization approaches. We also implemented efficient, offloaded 3D matrix multiply kernels for purification. (Section 6.)

In Section 7, we demonstrate results of large calculations on Stampede and Tianhe-2 supercomputers. Scalability is demonstrated on large molecular systems involving 2938 atoms and 27394 basis functions, utilizing 8100 nodes of Tianhe-2.

## 2 Background

In this section, we give the necessary background on the generic distributed Fock matrix construction algorithm needed for understanding the scalable implementation presented in this paper.

In each SCF iteration, the most computationally intensive part is the calculation of the Fock matrix,  $F$ . Each element is given by

$$F_{ij} = H_{ij}^{core} + \sum_{kl} D_{kl} (2(ij|kl) - (ik|jl)) \quad (1)$$

where  $H^{core}$  is a fixed matrix,  $D$  is the density matrix, which is fixed per iteration, and the quantity  $(ij|kl)$  is standard notation that denotes an entry in a four dimensional tensor of size  $n \times n \times n \times n$  and indexed by  $i, j, k, l$ , where  $n$  is the number of basis functions. To implement efficient HF procedures, it is essential to exploit the structure of this tensor. Each  $(ij|kl)$  is an integral, called an electron repulsion integral (ERI), given by

$$(ij|kl) = \int \phi_i(x_1)\phi_j(x_1)r_{12}^{-1}\phi_k(x_2)\phi_l(x_2)dx_1dx_2, \quad (2)$$

where the  $\phi$  functions are basis functions,  $x_1$  and  $x_2$  are coordinates in  $\mathbb{R}^3$ , and  $r_{12} = \|x_1 - x_2\|$ . From this formula, it is evident that the ERI tensor has 8-way symmetry, since  $(ij|kl) = (ij|lk) = (ji|kl) = (ji|lk) = (kl|ij) = (kl|ji) = (lk|ij) = (lk|ji)$ .

The basis functions are Gaussians (or combinations of Gaussians) whose centers are at the coordinates of one of the atoms in the molecular system. The formula (2) also shows that some values of  $(ij|kl)$  can be very small, e.g., when  $\phi_i(x)$  and  $\phi_j(x)$  are basis functions with far apart centers, then the product of these basis functions is small over all  $x$ . Small ERIs are not computed, which is an essential optimization in quantum chemistry called *screening*. To determine if an integral is small without first computing it, one uses the Cauchy-Schwarz relation

$$(ij|kl) \leq \sqrt{(ij|ij)(kl|kl)} \quad (3)$$

which gives a cheap-to-compute upper bound, once all possible values of  $(ij|ij)$  are precomputed and stored in a 2D array. If this upper bound is less than the screening tolerance  $\tau$ , then the integral is neglected.

To understand the computation of ERIs, we must know that basis functions are grouped into *shells*, which vary in size. All basis functions in a shell have the same atomic center. For efficiency, ERIs must be computed in batches called *shell quartets*, defined as

$$(MN|PQ) = \{(ij|kl) \text{ s.t. } i \in \text{shell } M, j \in \text{shell } N, \\ k \in \text{shell } P, l \in \text{shell } Q\}.$$

These batches are 4-dimensional arrays of different sizes and shapes. To apply screening to shell quartets, define the *screening value* for the pair of shell indices  $M$  and  $N$  as

$$\sigma(M, N) = \max_{i \in M, j \in N} (ij|ij).$$

Then the shell quartet  $(MN|PQ)$  can be neglected if

$$\sqrt{\sigma(M, N)\sigma(P, Q)} \leq \tau. \tag{4}$$

The generic distributed Fock matrix construction algorithm is shown as Algorithm 2. The algorithm loops over shell quartets rather than Fock matrix entries in order to apply screening to shell quartets, and to not repeat the computation of symmetric but otherwise identical shell quartets. Because of symmetry, each uniquely computed shell quartet contributes to the computation of 6 submatrices of  $F$  and requires 6 submatrices of  $D$ . The data access pattern arises from the structure of (1); see [5] for an explanation. The Fock and density matrices are partitioned and distributed onto nodes. Thus, inter-node communication is generally needed to receive submatrices of  $D$  and send submatrices of  $F$ . In NWChem and our own code, Global Arrays is used for storing  $D$  and  $F$  and implicitly managing communication of these submatrices.

---

**Algorithm 2:** Generic distributed Fock matrix construction

---

```

1 for unique shell quartets  $(MN|PQ)$  do
2   if  $(MN|PQ)$  is not screened out then
3     Compute shell quartet  $(MN|PQ)$ 
4     Receive submatrices  $D_{MN}, D_{PQ}, D_{NP}, D_{MQ}, D_{NQ}, D_{MP}$ 
5     Compute contributions to submatrices  $F_{MN}, F_{PQ}, F_{NP}, F_{MQ}, F_{NQ}, F_{MP}$ 
6     Send submatrices of  $F$  to their owners
7   end
8 end

```

---

### 3 Scalable Fock Matrix Construction

There are two main ways to parallelize Algorithm 2. The first approach is to use a dynamic scheduling algorithm to schedule tasks onto nodes, where each task is lines 3–6 in the algorithm for one or more shell quartets. This approach is naturally load balanced and is used by NWChem. The second approach is to statically partition the shell quartets among the nodes; each node executes lines 3–6 for the shell quartets in its partition. The advantage of this approach is that the necessary communication is known before computation starts, so the partitions can be chosen to reduce communication. Further, the required submatrices of  $D$  can be prefetched before computation, and submatrices of  $F$  can be accumulated locally before sending them all at once at the end of the computation. The disadvantage of the second approach, however, is that load balance and reducing communication are difficult to achieve.

In our previous work [9], we proposed a hybrid of the above two approaches. A static partitioning is used that reduces communication but only approximately balances the load across the nodes. A dynamic work stealing phase is used to polish the load balance. Specifically, when a node finishes its allocated work, it steals work from another node. The results of this approach showed better scalability than NWChem on the Lonestar supercomputer up to 343 nodes (3888 cores) [9]. However, when scaling up this technique to many more nodes on Tianhe-2, the static partitionings across more nodes were relatively less balanced, and the work stealing phase of the hybrid approach became a bottleneck due to the communication that it involves. To address these issues, we first improved the initial static partitioning, which also reduces the imbalance that needs to be handled in the work stealing phase. We next improved the work stealing phase by reducing the synchronization requirements. The improvements to the hybrid approach that were necessary for scalability are described in the following subsections.

### 3.1 Static partitioning

The static partitioning is a partitioning of the shell quartets among the nodes. To balance load, each partition should have approximately the same number of non-screened shell quartets (shell quartets surviving screening). To reduce communication, the computations with the non-screened shell quartets within a partition should share submatrices of  $D$  and  $F$  as much as possible.

We now present a framework for defining different partitionings of shell quartets. From Section 2, a shell quartet  $(MN|PQ)$  is indexed by four shell indices,  $M$ ,  $N$ ,  $P$ , and  $Q$ , which range from 1 to  $n_{\text{shells}}$ , where  $n_{\text{shells}}$  is the number of shells. Thus the shell quartets can logically be arranged in a 4D array. This array can be partitioned using 1D, 2D, 3D, or 4D partitionings. (More general ways of partitioning the 4D array, using irregularly shaped partitions, require listing all the shell quartets in a partition, but this is impractical due to the large number of shell quartets.)

A 1D partitioning is a partitioning of the indices along one of the dimensions; a 2D partitioning is a partitioning of the indices along two of the dimensions, etc. More precisely, a 1D partitioning into  $p$  parts is the  $p$  sets

$$(\mathcal{M}_i, : | :, :) \equiv \{(MN|PQ), \text{ s.t. } M \in \mathcal{M}_i, \text{ for all } N, P, Q\}, \quad i \in \{1, \dots, p\}$$

where  $\mathcal{M}_i$  is a subset of the shell indices  $\mathcal{I} = \{1, \dots, n_{\text{shells}}\}$  such that no two distinct subsets intersect, and the union of all subsets comprises all the indices  $\mathcal{I}$ . The above is a partitioning along the first dimension, and different “types” of partitionings arise from partitioning along different dimensions. However, due to 8-way symmetry in the 4D array of shell quartets (due to the 8-way symmetry in the 4D array of ERIs), partitionings along other dimensions are equivalent, e.g.,  $(\mathcal{M}_i, : | :, :)$  and  $(:, \mathcal{M}_i | :, :)$  contain the same shell quartets.

There are two different types of 2D partitionings into  $p = p_r \times p_c$  parts:

$$(\mathcal{M}_i, \mathcal{N}_j | :, :) \equiv \{(MN|PQ), \text{ s.t. } M \in \mathcal{M}_i, N \in \mathcal{N}_j, \text{ for all } P, Q\}, \quad i \in \{1, \dots, p_r\}, j \in \{1, \dots, p_c\}$$

and

$$(\mathcal{M}_i, : | \mathcal{P}_k, :) \equiv \{(MN|PQ), \text{ s.t. } M \in \mathcal{M}_i, P \in \mathcal{P}_k, \text{ for all } N, Q\}, \quad i \in \{1, \dots, p_r\}, j \in \{1, \dots, p_c\}$$

where  $\mathcal{M}_i$ ,  $\mathcal{N}_j$ , and  $\mathcal{P}_k$  are subsets of  $\mathcal{I}$ . Due to symmetry, other ways of partitioning along two dimensions are equivalent to one of the two types of partitioning above. In addition, all 3D partitionings are equivalent to each other, and there is only one way to define a 4D partitioning. In summary, there are five distinct forms of partitionings to consider, one each for 1D, 3D, and 4D, and two for 2D.

A specific partitioning is defined by how the indices  $\mathcal{I}$  are partitioned for each dimension, e.g., in the 1D case, this is how the subsets  $\mathcal{M}_i$  are chosen. The straightforward technique is to choose each subset

of  $\mathcal{I}$  to contain approximately the same number of shell indices, to attempt to balance load. To reduce communication of  $D$  and  $F$  submatrices, subsets should contain shell indices corresponding to geometrically nearby atomic centers. Both of these can be accomplished simply by ordering shell indices using a space-filling curve to promote geometric locality, and then partitioning the shell indices equally along this curve. Each partition is a set of shell quartets with similar combinations of shell indices, which reduces the number of distinct submatrices of  $D$  and  $F$  that are needed by the partition.

The above technique guarantees that the number of shell quartets before screening is about the same in each partition. Although there is no guarantee that the number of non-screened shell quartets in each partition will be similar, the hope is that due to the large number of shell quartets, the number of non-screened shell quartets in each partition will “average out.” To check this, and to check the communication cost, we measured properties of different forms of partitioning for two different test models. Table 1 shows results for a truncated globular protein model and Table 2 shows results for a linear alkane. In the tables, 2D, 3D, and 4D partitionings are shown with how many subsets were used along each dimension that is partitioned (e.g., 8x8 in the 2D case). The shell quartet balance is the ratio of the largest number of non-screened shell quartets in a partition to the average number in a partition. The communication balance is the maximum number of bytes transferred by a node to the average number of bytes transferred by a node. The average communication per node is reported in terms of the number of  $D$  and  $F$  submatrices and in kilobytes (kB). The last line in each table below the horizontal line is for a modified 2D partitioning that will be explained later.

We note that in these tables, we do not show results for 1D partitioning. Such partitionings limit the amount of parallelism that can be used. For example, if there are 1000 shells, parallelism is limited to 1000 cores. Due to this limitation, we do not consider 1D partitionings further in this paper.

The results in the tables suggest that 2D partitionings of the form  $(\mathcal{M}_i, : | \mathcal{P}_k, :)$  have the best load and communication balance. However, 3D and 4D partitionings require less average communication per node. In our previous work, 2D partitionings of the form  $(\mathcal{M}_i, : | \mathcal{P}_k, :)$  were used [9]. We continue to use this choice as Fock matrix construction is generally dominated by computation rather than communication, and thus it is more important to choose the option that balances load rather than the one that has the least communication [9]. In general, however, the best form of partitioning to use may depend on the relative cost of communication and computation, the size of the model problem, and the amount of parallel resources available.

Table 1: Characteristics of different forms of partitioning for 64 nodes. The test model is a truncated protein with 64 atoms, 291 shells, and 617 basis functions.

		shell quartet balance	communication balance	ave comm/node (submatrices)	ave comm/node (kB)
$(\mathcal{M}_i, \mathcal{N}_j   :, :)$	8x8	3.81	1.36	40731.3	633.6
$(\mathcal{M}_i, :   \mathcal{P}_k, :)$	8x8	2.01	1.57	42789.4	665.7
$(\mathcal{M}_i, \mathcal{N}_j   \mathcal{P}_k, :)$	4x4x4	7.71	4.71	18093.6	281.5
$(\mathcal{M}_i, \mathcal{N}_j   \mathcal{P}_k, \mathcal{Q}_l)$	4x4x2x2	11.38	4.98	16629.7	258.7
$(\mathcal{M}_i, :   \mathcal{P}_k, :)$	8x8	1.15	1.37	35674.4	555.0

In attempting to scale up the above technique on Tianhe-2, using partitions of the form  $(\mathcal{M}_i, : | \mathcal{P}_k, :)$ , we found poor load balance because the average number of shell quartets per partition is relatively small when the number of partitions is very large. This increases the standard deviation of the number of non-screened shell quartets in a partition. We thus sought a new partitioning with better load balance. The technique described above divides the indices equally along each dimension. Instead, we now seek to divide the indices possibly unequally such that the number of non-screened shell quartets is better balanced. For partitions of the form  $(\mathcal{M}_i, : | \mathcal{P}_k, :)$ , to determine the subsets  $\mathcal{M}_i$  and  $\mathcal{P}_k$ , we need to know the number of

Table 2: Characteristics of different forms of partitioning for 64 nodes. The test model is an alkane with 242 atoms, 966 shells, and 1930 basis functions.

		shell quartet balance	communication balance	ave comm/node (submatrices)	ave comm/node (kB)
$(\mathcal{M}_i, \mathcal{N}_j   :, :)$	8x8	6.37	3.09	82484.9	1283.1
$(\mathcal{M}_i, :   \mathcal{P}_k, :)$	8x8	1.07	1.25	84447.5	1313.7
$(\mathcal{M}_i, \mathcal{N}_j   \mathcal{P}_k, :)$	4x4x4	4.26	4.79	40707.0	633.3
$(\mathcal{M}_i, \mathcal{N}_j   \mathcal{P}_k, \mathcal{Q}_l)$	4x4x2x2	8.01	7.32	37533.6	583.9
$(\mathcal{M}_i, :   \mathcal{P}_k, :)$	8x8	1.03	1.20	80027.0	1245.0

non-screened shell quartets in the 2D *slices* of the 4D array of shell quartets,

$$(M, : | P, :) \equiv \{(MN | PQ) \text{ for all } N, Q\}, \quad M \in \{1, \dots, n_{\text{shells}}\}, \quad P \in \{1, \dots, n_{\text{shells}}\}.$$

The set of shell quartets in a 2D slice is called a *task*, which is the unit of work used by the work stealing dynamic scheduler to be described later. These  $(n_{\text{shells}})^2$  numbers can be stored in a 2D array (indexed by  $M$  and  $P$ ) and a better balanced 2D partitioning can be found. In the 4D case, this procedure is impossible because  $(n_{\text{shells}})^4$  is generally too large. In the 3D case, this procedure is only possible for small model problems where  $(n_{\text{shells}})^3$  is not too large. This leaves the two 2D cases as practical possibilities. We choose partitions of the form  $(\mathcal{M}_i, : | \mathcal{P}_k, :)$  rather than  $(\mathcal{M}_i, \mathcal{N}_j | :, :)$  because they are more balanced and require less communication as shown in the previous tables. Indeed, slices of the form  $(M, N | :, :)$  may contain widely varying numbers of non-screened shell quartets: if  $M$  and  $N$  are the indices of two shells with far-apart centers, then there will not be any shell quartets in  $(M, N | :, :)$  that survive screening.

Instead of counting the non-screened shell quartets in each of the above 2D slices, these counts can be estimated as follows, beginning with a few definitions. We say that the shell pair  $M, N$  is *significant* if

$$\sigma(M, N) \geq \tau^2 / m^*, \quad m^* = \max_{P, Q} \sigma(P, Q)$$

which is based on rearranging (4). We define the *significant set* of a shell as

$$\Phi(M) = \{N \text{ s.t. } \sigma(M, N) \geq \tau^2 / m^*\}$$

and  $\eta(M)$  to be the number of elements in  $\Phi(M)$ . From these definitions, an upper bound on the number of shell quartets in  $(M, : | P, :)$  that survive screening is  $\eta(M)\eta(P)$ . Figure 2 shows that the upper bound is a good estimate on the actual number of non-screened shell quartets in a slice  $(M, : | P, :)$ .

Now that the number of non-screened shell quartets can be estimated as the product of two terms, the slices can be assigned to partitions in a balanced manner very elegantly as we now describe. For  $p$  nodes (with  $p$  square), the assignment of slices to  $p$  partitions can be performed by dividing the *shells* into  $\sqrt{p}$  subsets, with the  $i$ th subset denoted as  $\mathcal{G}_i$ . We choose the subsets such that the sum of the  $\eta(M)$  for each subset is about the same,

$$\sum_{M \in \mathcal{G}_i} \eta(M) \approx \eta^*$$

where  $\eta^* = (\sum \eta(M)) / \sqrt{p}$ . The partitions or nodes can be indexed by  $(i, j)$  with  $i$  and  $j$  running from 1 to  $\sqrt{p}$ . Then slice  $(M, : | P, :)$  is assigned to partition  $(i, j)$  if  $M \in \mathcal{G}_i$  and  $P \in \mathcal{G}_j$ . The estimated number of shell quartets in each partition  $(i, j)$  is

$$\sum_{\substack{M \in \mathcal{G}_i \\ P \in \mathcal{G}_j}} \eta(M)\eta(P) = \sum_{M \in \mathcal{G}_i} \eta(M) \times \sum_{P \in \mathcal{G}_j} \eta(P) \approx (\eta^*)^2$$

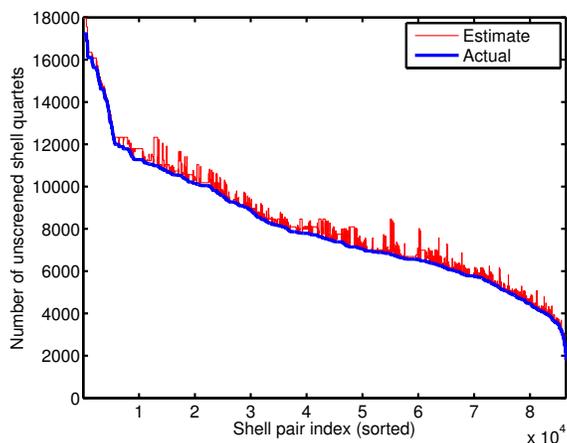


Figure 2: Number of shell quartets in each shell pair after screening, sorted by decreasing number of shell quartets. The graph shows that the estimate is a good upper bound on the actual number. The molecular model is the alkane  $C_{24}H_{50}$  with the cc-pVDZ basis set, giving 294 shells and 586 basis functions. The bound would be looser for larger molecules.

which is thus approximately balanced.

In Tables 1 and 2, the last row shows the result of this new partitioning technique. In both model problems, load balance is improved. As a side effect, the average communication cost is also slightly reduced.

We note in passing that NWChem uses a 4D partitioning of the ERI tensor into a large number of “tasks.” There is no attempt to balance the number of non-screened shell quartets in each task because dynamic scheduling of the tasks is used. There is also no attempt to schedule the tasks such that submatrices of  $D$  and  $F$  are reused as much as possible.

### 3.2 Work stealing

Static partitioning schemes for distributed Fock matrix construction have the advantage of being able to partition work such that communication is reduced. However, as the actual amount of work performed is difficult to predict, these schemes do not give perfect load balance in practice. Our approach is to combine static partitioning with work stealing. Nodes steal tasks from other nodes when they run out of tasks in their originally assigned partitions. This reduces the overall execution time by utilizing otherwise idle processors and essentially balancing the load.

Work stealing is an established paradigm for load balancing, particularly for shared memory systems where synchronization costs are relatively low. A steal operation has significant overhead, involving locking the work queue of another node to check for available work; if no work is found (a *failed steal*), another node must be checked. Dinan et al. [10] discuss the efficient implementation of work stealing on large distributed systems, primarily focusing on low level optimizations, such as managing the work queue and locking. Nikodem et al. [11] use work stealing to construct the Fock matrix, starting with a primitive partitioning of work. As steal operations have overhead, the task granularity cannot be too small, but tasks must be small enough so that all nodes can finish at approximately the same time. Nikodem et al. [11] discuss an optimization that sorts work by their granularity, so that larger tasks are performed first.

In previous work [9], we used a completely decentralized work stealing scheme where each node has its own task queue, to allow efficient local access, as suggested in Dinan et al. [10]. Victim nodes were chosen by scanning the logical 2D process grid from left-to-right and top-to-bottom starting from the thief

node. Although the standard strategy is to select the victim randomly, which has good provable properties [12], our strategy attempted to find work from a nearby node that might involve similar sets of  $D$  and  $F$  submatrices. However, when scaling up this work stealing implementation on Tianhe-2 using many more nodes than we had used previously, we encountered a very large number of failed steals, adding significantly to the overhead. This can happen if the load is very unbalanced, as can happen near the end of a computation when some nodes still have work, but many nodes have exhausted their work.

To address the large number of failed steals on Tianhe-2, we designed a *hierarchical* work stealing scheme. The main idea is to use a shared variable synchronized across all nodes (implemented using Global Arrays) that indicates whether any node in a group of nodes has work. Each node still has its own work queue. However, it is now possible to tell in a single operation if an entire group of nodes does not have any work, to reduce the number of failed steals. Pseudocode for this hierarchical (two-level) work stealing scheme is shown in Algorithm 3.

In the algorithm, the nodes or processes are assumed to be arranged in a logical  $p_r \times p_c$  process grid. The nodes are grouped by row, that is, the process row  $l$  forms the process group  $PG_l$ . The global array  $W$  indicates which groups still have tasks. Initially all entries of  $W$  are set to 1, and  $W_l = 1$  means  $PG_l$  still has remaining tasks. The algorithm terminates when all  $W_l$  are zero. The variable  $C$  acts as a node’s private counter that is used to determine when all nodes in its victim’s process group have no more tasks to steal. Within a group, the victim is selected randomly. If the victim has tasks to steal, then the thief steals *half* of the victim’s tasks. This is the classic choice, which balances the work between the victim and the thief.

---

**Algorithm 3:** Hierarchical work stealing dynamic scheduling (see text for explanation of variables).

---

```

1 On a node in  $PG_k$  do
2 while there are entries of  $W$  equal to 1 do
3   Select a victim process group  $PG_l$  with  $W_l = 1$  that is closest to  $PG_k$ 
4    $C \leftarrow \emptyset$ 
5   repeat
6     Randomly select a node  $n$  from  $PG_l$ 
7     if  $n \notin C$  then
8       while the task queue of  $n$  has tasks do
9         steal half of the tasks from  $n$ 
10      end
11       $C \leftarrow C \cup n$ 
12    end
13  until  $|C| = p_c$ 
14   $W_l \leftarrow 0$ 
15 end

```

---

To understand the effect of combining static partitioning with work stealing dynamic scheduling, Table 3 shows the load balance ratio for a test protein-ligand system. The load balance ratio is the ratio of the maximum compute time among all nodes to the average time for the ERI calculations and the local updates of the Fock matrix. The table shows that for static partitioning alone, load imbalance increases with number of nodes, but this is ameliorated by work stealing, which limits the imbalance to about 5%. Further, Table 4 shows the number of steals that were performed for different molecular problem sizes and for different numbers of nodes. For smaller problems and for more nodes, there are more steals because the static partitioning is less balanced.

Table 3: Fock matrix construction load balance ratio for static partitioning alone and hybrid of static partitioning and work stealing. Test system is a protein-ligand system, 1hsg\_28, with 122 atoms and 1159 basis functions.

Nodes	static	hybrid
1	1.000	1.000
9	1.077	1.049
36	1.183	1.048
64	1.293	1.046
144	1.358	1.051
225	1.439	1.046
529	1.457	1.047

Table 4: Number of steals in the work stealing phase of distributed Fock matrix construction. The test systems 1hsg\_28, 1hsg\_38, 1hsg\_45, 1hsg\_90 have 1159, 3555, 5065, 11163 basis functions, respectively.

Nodes	1hsg_28	1hsg_38	1hsg_45	1hsg_90
1	0	0	0	-
9	2	2	2	-
36	34	18	14	6
64	62	32	24	9
144	134	87	65	42
225	269	160	127	109
529	2444	380	366	349
1024	-	1738	1310	797

### 3.3 Global Arrays on Tianhe-2

Our HF code, like NWChem, uses Global Arrays [13] for communication and synchronization between nodes. Thus the performance of Global Arrays on Tianhe-2 is critical for scalability of our code. The communication runtime of Global Arrays is called ARMCI, which supports Put (write), Get (read), and Accumulate (update) operations, as well as atomic operations on single integers. On many platforms, ARMCI is implemented directly on top of a system-specific interface, e.g., Verbs for InfiniBand and DCMF for Blue Gene/P. However, there is no implementation for the Galaxy Express network on Tianhe-2. Instead, we had to use the portable ARMCI-MPI implementation which is built on MPI one-sided communication (a.k.a. remote memory access, or RMA) as a portable conduit [14]. ARMCI-MPI originally targeted the MPI-2 RMA features, which was sufficient to run NWChem across a number of supercomputing platforms [14], but the performance was limited due to the absence of atomics and nonblocking operations in MPI-2 RMA.

To improve performance of our HF code on Tianhe-2, we optimized the implementation of ARMCI-MPI for MPI-3 RMA, including a rewrite of the nonblocking ARMCI primitives. Many of the new features in MPI-3 RMA were critical for this optimized version of ARMCI-MPI, including window allocation (e.g., `MPI_Win_allocate`), atomics (`MPI_Fetch_and_op`) [15], unified memory, and better synchronization (`MPI_Win_lock_all` and `MPI_Win_flush`). Additionally, we added explicit progress calls inside the library to ensure responsive remote progress, which is critical for remote accumulate operations, among others. Thread-based asynchronous progress was found to be ineffective, due to the overhead of mutual exclusion between the communication thread and the application (main) thread. Finally, we found that it was necessary to use an existing feature in ARMCI-MPI to work around a bug in the datatypes implementation of MPI on Tianhe-2 by communicating multiple contiguous vectors rather than a single subarray.

The work stealing dynamic scheduler uses `MPI_Fetch_and_op` operations. While our code scales very well on both Lonestar and Stampede up to 1000 nodes, the performance is relatively poor on Tianhe-2 even

for 256 nodes. Efficient `MPI_Fetch_and_op` operations rely on hardware support of remote atomics. However, this is not supported by Tianhe-2, resulting in poor `MPI_Fetch_and_op` performance on this machine. This issue was our main motivation for improving the quality of the static partitioning, in order to reduce the number of steals needed in the work stealing dynamic scheduler. We note that this issue is much more serious for NWChem, which uses a globally shared counter (the `NXTVAL` operation in Global Arrays) every time a task is acquired.

#### 4 Optimization of Integral Calculations

In Fock matrix construction, the vast majority of the execution time is spent computing ERIs. Thus much effort has been devoted by researchers to develop efficient implementations for ERIs, including on Intel Xeon Phi [16], GPUs [17, 18, 19, 20, 21, 22], and specialized hardware [23, 24]. Early GPU implementations for quantum chemistry simply offload integral calculations to the GPU, e.g., [19]. This, however, entails transferring ERIs from the GPU to the host, which is a very large communication volume relative to computation. Later implementations compute the Fock matrix on the GPU to avoid this large data transfer, e.g., [22]. Some previous GPU implementations were also concerned with precision. Because earlier generations of GPU hardware greatly favored single precision calculation, but ERIs must be calculated in double precision, some of these GPU implementations considered mixed-precision calculations, where single precision is used when possible and double precision is used when necessary [17, 21]. The double precision performance of current GPUs make some of these techniques unnecessary. We also note that current GPU implementations generally assume single node computations, to simplify the grouping of shells of the same type, to reduce thread divergence in the calculations.

A notable limitation of most GPU implementations is that they do not port the full functionality of a typical integrals package, due to the complexity of ERI calculations and the wide range of possible ERIs to compute. A GPU implementation may only compute integrals involving certain shell types [18, 20], thus limiting the types of chemical systems that can be simulated. An advantage of Intel Xeon Phi is that the full functionality of an integrals package can at least be ported easily.

We used the ERD integral library [25] and optimized it for our target platforms, Ivy Bridge and Intel Xeon Phi, although we expect these optimizations to also benefit other modern CPUs. The ERD library uses the Rys quadrature method [26, 27] which has low memory requirements and is efficient for high angular momentum basis functions compared to other methods. The computation of a shell quartet of ERIs by Rys quadrature requires several steps, including computation of Rys quadrature roots and weights, computation of intermediate quantities called 2D integrals using recurrence relations, computation of the constants used in the recurrence relations (which depend on the parameters of the basis functions in each shell quartet), and computation of each ERI from the 2D integrals. Reuse of the 2D integrals in a shell quartet is what requires ERI calculations to be performed at least one shell quartet at a time. Due to the many steps involved in computing ERIs, there is no one single kernel that consumes the bulk of the time. Each step consumes a small percentage of the total time (Figure 3). We thus had a considerably large optimization effort, encompassing nearly 30000 lines of Fortran code.

Loops were restructured so that they could be vectorized and exploit wide SIMD units. For example, the “vertical” recurrence relations can be computed in parallel for different quadrature roots and different exponents. To increase developer productivity, we relied on programmer-assisted compiler auto-vectorization for several parts of the ERI code. We aligned most arrays on the width of SIMD registers and padded them to be a multiple of the SIMD register size. We annotated pointer arguments with the `restrict` keyword, and used Intel compiler-specific intrinsics `__assume_aligned` and `__assume` to convey information about array alignment and restrictions on variable values. We also employed `#pragma simd` to force loop vectorization. In addition, we explicitly vectorized several hot-spots using intrinsics where the compiler lacked high-level

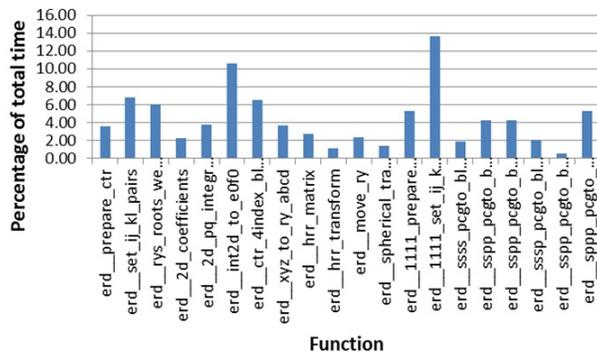


Figure 3: Breakup of runtime for the original ERD code.

information to generate optimal code. Wherever possible and beneficial, we tried to merge multiple loop levels to get a larger number of loop iterations, thereby achieving better SIMD efficiency. In almost every function, we also had to work around branching inside loops, in order to fully vectorize the code. We note, however, that in ERI calculations, there are many scalar operations and the loops generally have a relatively small number of iterations, limiting the performance gains on SIMD hardware.

In the process of converting the baseline Fortran code to C99, we found several low-level optimizations to be beneficial. First, instead of using globally allocated scratch arrays, we used local on-stack variable-length arrays provided by the C99 standard. Second, we converted signed 32-bit indices to unsigned 32-bit indices because, on x86-64, the processor needs to extend a signed 32-bit to 64 bits to use it as an index for a memory load or store. For unsigned indices, an extension instruction is not necessary because any operation on the low 32 bits implicitly zeroes the high part. Third, we found register pressure to be a problem. The ERD code was originally developed for the PowerPC architecture, which has 32 general purpose and 32 floating-point registers. On x86-64 we have only 16 general-purpose and 16 floating-point/SIMD registers (32 SIMD registers on Intel Xeon Phi). In our optimized code we revised interfaces to reduce the number of function parameters, and thus lowered the register pressure.

We observed that 30% of ERD computation time is devoted to primitive screening. This operation computes an upper bound on the size of the integrals in a shell quartet. If this upper bound is below a threshold, then the shell quartet is not computed. The bound, however, requires division and square root operations, which are not pipelined on the CPU and which require several multiply-add instructions on Intel Xeon Phi. Computation of this bound was rearranged to avoid these operations. Furthermore, the bound also requires computing the Boys function. The zero-order Boys function is defined as

$$F_0(x) = \int_0^1 \exp(-t^2 x) dt = \begin{cases} \frac{\sqrt{\pi}}{2} \frac{\operatorname{erf}\sqrt{x}}{\sqrt{x}} & \text{if } x > 0 \\ 1 & \text{if } x = 0. \end{cases}$$

The original ERD approximates the Boys function via a set of Taylor expansions on a grid of points. Such an approximation, however, is suboptimal for the following reasons. First, although Taylor expansions give good approximations in the vicinity of tabulated points, they are much less accurate away from tabulated points. Second, fetching coefficients at tabulated points in vectorized code requires a gather operation, which is lacking in Ivy Bridge processors and is expensive, albeit supported, on Intel Xeon Phi coprocessors. We derived a more efficient way to compute the Boys function based on Chebyshev polynomials, which can minimize the maximum error over an interval. The main idea is to manipulate the bound so that we need to approximate  $(\operatorname{erf}\sqrt{x})^2$ . This can be well approximated by a 5th-degree Chebyshev polynomial approximation.

Table 5 shows the performance improvement factors due to our ERI optimizations. Since ERI calculations run independently on each node, these tests were performed using a single node (24 threads for Ivy Bridge and 224 threads for Xeon Phi). The overall improvement factor averages 2.3 on Ivy Bridge and 3.1 on Xeon Phi. From timings with vector instructions turned off, we believe the greater improvement for Intel Xeon Phi is due to better SIMD usage. However, SIMD usage is still low due to short loops. In the tests, four very different molecular systems were used, ranging from 936 to 1424 atoms and using the cc-pVDZ basis set: a linear alkane, a 19mer segment of DNA, a planar graphene molecule, and a truncated globular protein-ligand system (1hsg). The different molecular configurations affect the screening procedures used in the integral code.

Table 5: ERI calculation performance improvement factor of the optimized code over the original code.

	Dual Ivy Bridge	Intel Xeon Phi
alkane_1202	2.32	3.13
dna_19mer	2.28	3.20
graphene_936	2.17	2.98
1hsg_100	2.44	3.25

## 5 Heterogeneous Computation

Each node of Tianhe-2 contains 2 Xeon CPUs and 3 Intel Xeon Phi coprocessors. There are two main modes of using the coprocessors simultaneously with CPUs in a distributed computation: 1) offload mode, where CPU processes offload work to be run on coprocessors, and 2) symmetric mode, where MPI processes run on both CPUs and coprocessors, and communication between processes fundamentally uses some form of message passing. We did not choose symmetric mode because CPU processes and coprocessor processes have vastly different capabilities in terms of main memory available and interprocess communication performance, making an already challenging load balancing problem even more challenging. We thus chose offload mode for heterogeneous computations, with one MPI process per node.

To load balance Fock matrix construction between CPUs and coprocessors, we again used a work stealing scheduler, this time at the node level. A special thread running on a dedicated CPU core is responsible for offloading tasks onto coprocessors and for managing the task queues of the coprocessors. The tasks for a node are initially partitioned among the threads on the CPUs. Threads, including the special thread, steal from each other when they run out of work.

Our heterogeneous implementation computes Fock submatrices on both CPUs and coprocessors simultaneously. In particular, by offloading Fock matrix construction rather than simply ERI calculations onto coprocessors, we only require transferring  $D$  and  $F$  submatrices across the PCIe interface. This is a much smaller volume of communication than transferring the ERIs if ERI calculations were offloaded. The special thread managing the coprocessors also is responsible for transferring the  $D$  and  $F$  submatrices between the host and coprocessors. The initial partitioning of nodes is similar to the distributed partitioning to maximize reuse of  $D$  and  $F$  submatrices, which is particularly important for partitions on coprocessors, due to limited memory on coprocessors.

Threads on a node may simultaneously want to update the same Fock submatrices. This can be done safely by using atomic operations. However, atomic operations have relatively low performance on Intel Xeon Phi due to the large number of cores. An alternative is for each thread to store its own copy of the Fock submatrices, and perform a reduce operation at the end. While this approach works very well for CPU-only computations, we generally do not have enough space to store a copy of the Fock submatrices for each of up to 224 threads on the coprocessors.

Our solution is to combine the use of atomic operations with multiple copies of Fock submatrices, as follows. First, it is unnecessary to store one Fock matrix copy for each Intel Xeon Phi thread. Four threads on a core can share one copy, as atomic operations within a core have low overhead. Second, for each task, there are six submatrices of  $F$  that need to be updated (See Algorithm 2). Not all of these blocks are the same size, due to different types of shells. Instead of storing copies of all of these blocks, we only store copies for the smaller blocks, and use atomic operations for single copies of the larger blocks. We found experimentally that this approach introduces less than 5% overhead due to atomic operations, while the memory requirement remains manageable.

To test the efficiency of offloading (which involves PCIe communication) and the heterogeneous work stealing scheduler (which attempts to balance the load), we measured the *offload efficiency*. For dual Ivy Bridge (IVB) and dual Intel Xeon Phi, the offload efficiency is defined as the ratio of two speedups: the actual speedup vs. theoretical speedup, where the actual speedup is the speedup of dual IVB with dual Phi over single IVB, and the theoretical speedup is the speedup if the dual IVB and dual Phi ran independently with no offload overheads. More precisely, if one Phi behaves like  $F$  IVB processors, then with two IVB processors and two Phi coprocessors, the theoretical speedup is  $(2 + 2F)$ . The quantity  $F$  may be measured as the ratio of the time consumed for one IVB processor vs. the time consumed by one Phi coprocessor for the same workload. Table 6 shows timings for Fock matrix construction for different node configurations and the offload efficiency. The timings use a single node, simulating one MPI process of a multi-node run. For different molecular systems, the offload efficiency is high, indicating little overhead due to offloading and dynamic scheduling involving the four processing components in this test. We note that the results show that the Intel Xeon Phi is slightly slower than one Intel Ivy Bridge processor (socket). The main reason for this is the poor use of SIMD in the integral calculations, as mentioned earlier.

Table 6: Speedup compared to single socket Ivy Bridge (IVB) processor, and offload efficiency for dual IVB and dual Intel Xeon Phi.

Molecule	single IVB	single Phi	dual IVB	dual IVB and dual Phi	Offload efficiency
alkane_1202	1	0.84	1.98	3.44	0.933
dna_19mer	1	0.98	2.00	3.75	0.945
graphene_936	1	0.96	2.00	3.71	0.944
1hsg_100	1	0.98	2.01	3.76	0.950

## 6 Density Matrix Calculation

Besides construction of the Fock matrix  $F$ , the other major step in each iteration of the SCF algorithm is the calculation of the density matrix  $D$ . In Hartree–Fock, this is traditionally performed using an eigendecomposition of  $F$ , computing  $D$  as

$$D = C_{occ} C_{occ}^T$$

where  $C_{occ}$  is the matrix formed by the lowest energy eigenvectors of  $F$ , corresponding to the number of occupied orbitals of the molecular system. Although massive parallel resources can be used to compute  $F$  of moderate size, the same resources cannot be efficiently employed to compute the eigenvectors of  $F$ , due to the relatively small workload and lack of large amounts of parallelism in this computation.

Our solution is to use a “diagonalization-free” method that avoids solving an eigenvalue problem and computes  $D$  directly from  $F$ . The method, in its most basic form, is known as McWeeny purification [28]. The algorithm is based on matrix multiplication, starting with an appropriate  $D_0$ ,

$$D_{k+1} = 3D_k^2 - 2D_k^3$$

and thus it can be very efficient on modern processors, including in distributed environments. We use a variant of McWeeny purification, called canonical purification [29], which allows us to compute  $D$  based on a given number of lowest energy eigenvalues (rather than the eigenvalue or chemical potential, in standard McWeeny purification). The purification iterations are stopped when  $\|D_k - D_k^2\|_F < 10^{-11}$ , i.e., when the approximation  $D_k$  is nearly idempotent.

For distributed matrix multiplication, we use a 3D algorithm [30, 31], which utilizes a 3D processor mesh, and which has asymptotically lower communication cost (by  $p^{1/6}$  for  $p$  nodes) than 2D algorithms such as SUMMA [32]. However, 3D algorithms require more memory and have the additional cost and complexity of redistributing the data to a 3D partitioning, which may originally be in a 2D partitioning.

Our implementation offloads local dgemm computations to Intel Xeon Phi coprocessors using the Intel MKL offload library. While we see up to  $6\times$  speedup due to offload to three coprocessors for large matrices, the speedup shrinks as the number of nodes increases and the local matrix size decreases. For very small matrices, offloading to coprocessors results in a slowdown. This is due to higher overhead of copying matrices over PCIe and lower dgemm efficiency for smaller matrices. In addition, for very large numbers of nodes, the overhead of communication dominates, and accelerating computation via offload has a very small impact on overall performance. Hence our implementation uses dgemm performance profiling information to dynamically decide when to offload to the coprocessors.

As shown earlier in Figure 1, density matrix purification can be much more scalable than eigendecomposition based approaches. For small numbers of nodes, however, eigendecomposition approaches are still faster.

## 7 Performance Results

### 7.1 Test setup

The performance of our HF code, called GTFock, is demonstrated using truncated protein-ligand systems. The base system is a drug molecule, indinavir, bound to a protein, human immunodeficiency virus (HIV) II protease. The entire protein-ligand complex (Figure 4, pdb code 1HSG) is too large to study quantum mechanically, so truncated models are used, where the ligand is simulated with residues of the protein within a truncation radius. We use different truncation radii to generate test systems of different sizes, to test the performance of our code for different problem sizes. In addition to studying code performance, studying sequences of larger problems like this has scientific importance in understanding the limits of truncated model systems. This particular protein-ligand complex has been studied earlier using Hartree-Fock and other methods, but here we are able to go to much larger model systems, with 2938 atoms compared to 323 atoms in previous work [33].

Table 7 lists a sample of the molecular systems of different sizes that were tested. The largest corresponds to all protein residues 18 Å from the ligand (the entire protein is within 22 Å). A test system named 1hsg\_35 indicates that all residues containing any atom within 3.5 Å of any ligand atom is included in the model. Bonds cut by the truncation are capped appropriately.

GTfock implements the SCF iteration of Algorithm 1 but uses purification rather than eigendecomposition for computing the density matrix,  $D$ . In addition, the GTfock code accelerates the convergence of the SCF iteration by using direct inversion of the iterative subspace (DIIS) [34]. In this method, an improved approximation to the Fock matrix is formed from a linear combination of Fock matrices from previous iterations. The linear combination is the one that reduces the error, as measured by the commutator  $(FDS - SDF)$ , where  $S$  is the overlap matrix. The additional computational cost is small relative to the cost of forming  $F$ , but the real cost is the need to store the Fock matrices from previous iterations, reducing memory available for other optimizations. GTfock also uses a standard initial guess for the density matrix called “superposition of atomic densities” (SAD). To verify accuracy of GTfock, results of smaller simula-

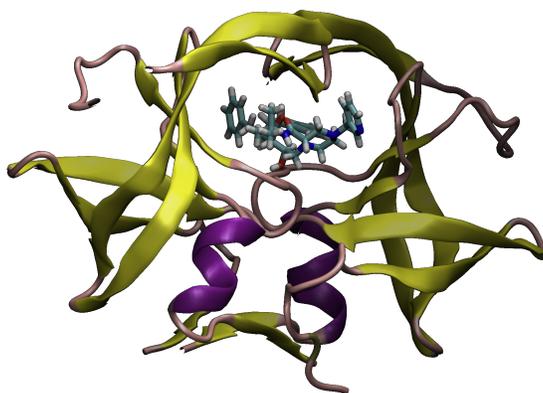


Figure 4: Indinavir bound to HIV-II protease (pdb code 1HSG).

Table 7: Test systems of varying size using the cc-pVDZ basis set (unoptimized contractions).

Molecule	Atoms	Shells	Basis Functions
1hsg_35	220	981	2063
1hsg_45	554	2427	5065
1hsg_70	789	3471	7257
1hsg_80	1035	4576	9584
1hsg_100	1424	6298	13194
1hsg_140	2145	9497	19903
1hsg_160	2633	11646	24394
1hsg_180	2938	13054	27394

tions were checked against those from the Q-Chem package [35]. Finally, we note that a Cauchy-Schwarz tolerance of  $\tau = 10^{-10}$  was used for screening ERIs.

Performance results were measured on the Tianhe-2 supercomputer located at the National Supercomputing Center in Guangzhou, China. The machine was developed by the National University of Defense Technology, China. Tianhe-2 is composed of 16000 nodes with a custom interconnect called TH Express-2 using a fat-tree topology. Each node is composed of two Intel Ivy Bridge E5-2692 processors (12 cores each at 2.2 GHz) and three Intel Xeon Phi 31S1P coprocessors (57 cores at 1.1 GHz). Memory on each node is 64 GB DRAM and 8 GB on each Intel Xeon Phi card. Capable of a peak performance of 54.9 PFlops, Tianhe-2 has achieved a sustained performance of 33.9 PFlops with a performance-per-watt of 1.9 GFlops/W. Tianhe-2 has 1.4 PB memory, 12.4 PB storage capacity, and power consumption of 17.8 MW. We were able to use 8100 nodes of Tianhe-2 for our tests.

Performance results were also measured on the Stampede supercomputer located at Texas Advanced Computing Center. We were able to use 1024 nodes of the machine, which is the limit for jobs on the “large” queue. We used nodes composed of two Intel Sandy Bridge E5-2680 processors (8 cores each at 2.7 GHz) with one Intel Xeon Phi coprocessor (61 core). Memory on these nodes is 32 GB DRAM and 8 GB for the Intel Xeon Phi card.

## 7.2 SCF strong scaling

Figure 5 shows timing and speedup results for a single SCF iteration for the 1hsg\_80 model on Stampede and Figure 6 shows the same for 1hsg\_180 on Tianhe-2. In the figures, “Total” denotes the total time for

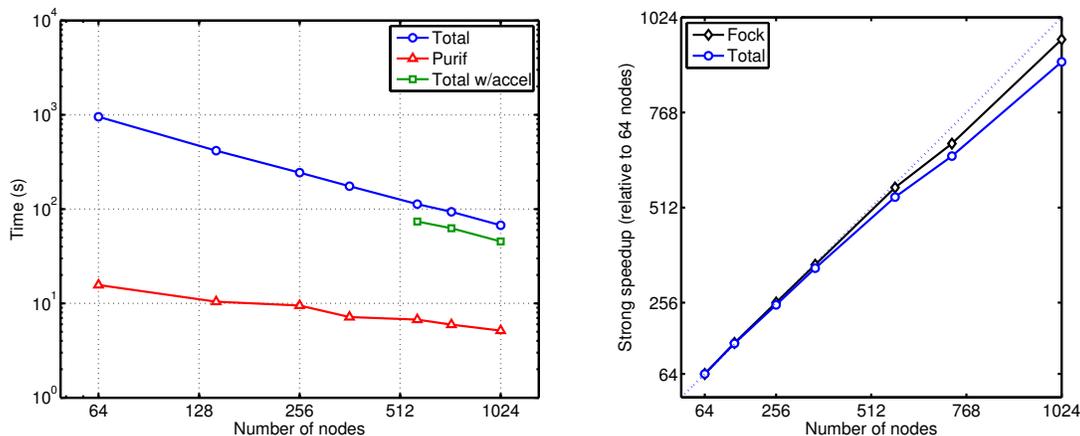


Figure 5: Timings and speedup for one SCF iteration for 1hsg\_80 (9584 basis functions) on Stampede.

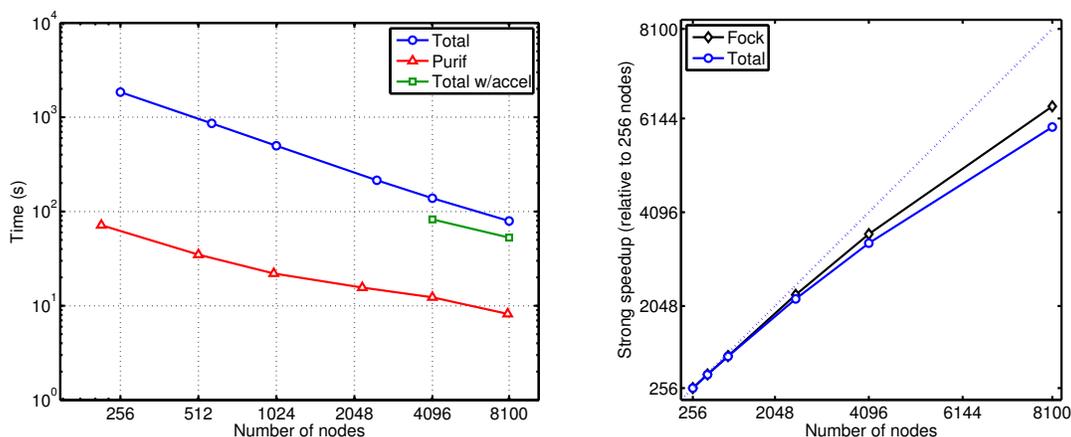


Figure 6: Timings and speedup for one SCF iteration for 1hsg\_180 (27394 basis functions) on Tianhe-2.

CPU-only computations, “Purif” denotes portion of the CPU-only time for canonical purification, and “Total w/accel” denotes total time for heterogeneous (CPU+coprocessor) computations. For the speedup graphs, “Fock” denotes the time for CPU-only Fock matrix construction. We chose the number of nodes of be squares, but this is not necessary for our code. For runs on Stampede, we used 2D matrix multiplication for purification. For runs on Tianhe-2, we used 3D matrix multiplication. In this case, purification used the nearest cubic number of nodes smaller than the number of nodes for Fock matrix construction.

The results show good speedup for Fock matrix construction. An important observation is that timings for purification are small relative to those for Fock matrix construction. Also important is the observation that the purification timings continue to decrease for increasing numbers of nodes. This is despite the fact that, as we increase the number of nodes, the dgemms performed by each node in the distributed matrix multiply algorithm become smaller and less efficient, while the communication cost increases. Due to the increased inefficiency, the scaling of purification is much poorer than the scaling of Fock matrix construction. However, since timings for purification remain relatively small, they make a relatively small impact on total speedup, as shown as the difference between total speedup and Fock matrix construction speedup. On Tianhe-2, CPU-only relative speedup at 8100 nodes is 5954.1, or 73.5% parallel efficiency.

The speedup of heterogeneous over CPU-only computations on Stampede is 1.49 to 1.53. On Tianhe-2, this speedup is 1.50 to 1.68. In heterogeneous mode, small numbers of nodes could not be used for large

problems, due to limited memory for submatrices of  $D$  and  $F$  on the coprocessors. To explain why, note that for small numbers of nodes, each node performs more tasks than when large numbers of nodes are used. When more tasks are performed, more submatrices of  $D$  must be prefetched and more submatrices of  $F$  must be stored locally. In heterogeneous mode, these submatrices of  $D$  and  $F$  are stored on each Intel Xeon Phi card on the node. With limited memory on the coprocessor cards, heterogeneous mode can only be used for large node counts. However, heterogeneous mode can be used for smaller node counts for smaller problems.

### 7.3 SCF weak scaling

Weak scaling is difficult to measure because it is difficult to increase computational work exactly proportionally with number of nodes. This is primarily because, due to ERI screening, the amount of computational work is not known beforehand. However, the ERI calculation time can be measured and used as a proxy for the amount of computational work, assuming the load is balanced. We timed SCF for a set of test systems of various sizes (Table 7), using a number of nodes approximately proportional to the square of the number of basis functions for each problem. To produce the plot for weak scaling, the timings are “corrected” by scaling them by the proxy ERI calculation time. Table 8 shows the timings for SCF using heterogeneous computations. “ERI” denotes the portion of Fock matrix construction time spent in ERI calculations. The resulting weak scaling plot is shown in Figure 7. Weak scaling for CPU-only computations is also plotted for comparison. As expected, scalability is better for CPU-only because computations are slower (no coprocessor acceleration) while communication remains the same. Table 8 shows that communication cost for Fock matrix construction (difference between “Fock” and “ERI” columns) increases with the number of nodes and becomes a substantial portion of the total time. Purification remains a relatively smaller portion of the total time for all problem sizes.

Table 8: Timing data (seconds) for one SCF iteration for different problem sizes using heterogeneous computations. “ERI” denotes the portion of Fock matrix construction time spent computing ERIs. Time for purification is also shown. We note that 33 to 36 purification iterations were used for converging the density matrix calculation.

Molecule	Nodes	Fock	ERI	Purif	Total
1hsg_35	64	7.3	5.9	0.7	8.4
1hsg_45	256	21.1	18.2	1.2	22.7
1hsg_70	576	25.5	19.2	1.5	27.3
1hsg_80	1024	30.0	19.4	1.9	32.3
1hsg_100	2304	31.9	21.3	3.0	35.2
1hsg_140	4096	38.3	25.1	5.0	43.6
1hsg_160	6400	41.4	25.5	5.7	47.5
1hsg_180	8100	44.1	26.2	8.4	52.9

### 7.4 Flop rate

Although SCF is not a flop-rich algorithm, it is still interesting to compute the achieved flop rate. We show this as an example of how to count flops in a complex calculation where the number of flops performed cannot be estimated analytically. We count the flops in purification and ERI calculation; all other operations (e.g., summation of the Fock matrix) perform a relatively small number of flops, which we neglect. The number of flops spent in purification can be counted analytically. ERI calculation, however, is very unstructured, and the different types of integrals and different ways for integrals to be screened out makes analytical counting unwieldy. Instead, we use hardware counters to measure the number of flops in ERI calculations.

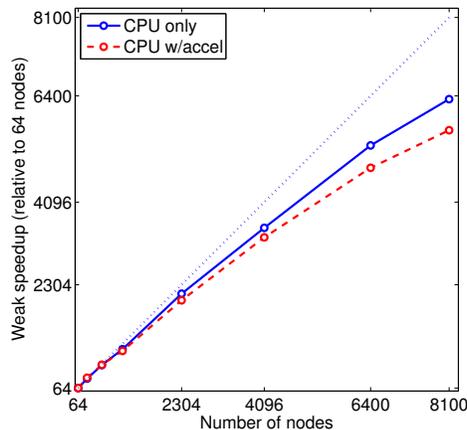


Figure 7: Weak speedup, relative to 64 nodes.

Specifically, we use the *perf\_events* interface exported by recent versions of the Linux kernel. As Intel Xeon Phi does not have proper flop counters support, we perform all hardware event measurements on x86 CPUs. We compiled with the *-no-vec* option to avoid inaccuracies due to partial use of vector registers in SIMD operations. Estimates of flop counts for ERI calculations are shown in Table 9 for a selection of test systems. The columns in the table are estimates computed the following ways:

1. Retired floating point operations on AMD Piledriver, which separately counts multiplications and additions, and jointly counts divisions and square roots. We call this count *CPU intrinsic*. We verified the total of these counts using the retired flops counter on Intel Harpertown.
2. Floating point operations at the execution stage of the pipeline on Intel Nehalem. The counters we use also count compares, and may also overcount due to speculative execution and recirculations of these operations in the pipeline. We call this count *CPU executed*, and it is an upper bound on the number of flops performed. This result gives additional confidence in our intrinsic count.
3. Intel Xeon Phi does not have counters for flops. Also, Intel Xeon Phi does not have single-instruction division and square root operations; these functions are computed with a sequence of multiplication, addition, and fused multiply-add operations. Square roots and divisions require a sequence of 10 and 11 flops, respectively, and thus the flop counts on Intel Xeon Phi are higher than on CPUs. We instrumented our code to count the number of square root operations, and used AMD Piledriver counts to deduce the number of divisions. We used these results to estimate the number of flops performed on Intel Xeon Phi.

Table 9: Flop counts (Gflops) for ERI calculation.

Molecule	CPU Intrinsic	CPU Executed	Intel Xeon Phi
1hsg_35	30,646	33,951	45,105
1hsg_45	386,695	448,561	575,323
1hsg_80	1,830,321	2,124,477	2,721,020
1hsg_140	8,751,659	10,223,033	13,027,801
1hsg_160	13,844,868	16,141,342	20,547,502
1hsg_180	17,820,050	20,853,142	26,487,829

Table 10 shows approximate flop rates using the timing data from the previous tables. Interestingly, purification, which is based on dgemm, has a lower rate than ERI calculation. This is because of the small size of the matrices per node, as well as communication costs. In summary, for the largest problem on 8100 nodes, the aggregate flop rate for HF-SCF is 441.9 Tflops/s.

Table 10: Flop rates (Tflops/s) for Table 8.

	Nodes	ERI	Purif	Total SCF
lhsg_35	64	6.6	1.7	4.9
lhsg_45	256	27.5	14.2	22.6
lhsg_70	576	62.3	34.6	44.9
lhsg_80	1024	121.7	63.3	75.2
lhsg_100	2304	230.0	107.3	142.2
lhsg_140	4096	450.0	222.4	264.7
lhsg_160	6400	701.3	356.2	383.6
lhsg_180	8100	879.2	352.6	441.9

## 8 Conclusions

The Hartree–Fock method has a complex data access pattern and irregular computations that are challenging to vectorize. This paper presented an optimized, scalable code for Hartree–Fock calculations. The software, called GTFock, has been released in open-source form at <https://code.google.com/p/gtfock>. The code described in the paper corresponds to version 0.1.0 of GTFock, except for some optimizations that were necessary for efficient performance on the Tianhe-2 proprietary interconnect (e.g., explicit MPI progress calls). The repository also contains the test molecular geometries used in this paper. GTFock can be integrated into existing quantum chemistry packages and can be used for experimentation as a benchmark for high-performance computing. The code is capable of separately computing the Coulomb and exchange matrices and can thus be used as a core routine in other quantum chemistry methods.

Scalability problems were encountered when scaling up the code to 8100 nodes on Tianhe-2. These were resolved by using a better static partitioning and a better work stealing algorithm than used in previous work. We also fully utilized the Intel Xeon Phi coprocessors on Tianhe-2 by using a dedicated thread on each node to manage offload to coprocessors and to use work stealing to dynamically balance the work between CPUs and coprocessors. The ERI calculations were also optimized for modern processors including Intel Xeon Phi.

The partitioning framework for Fock matrix construction presented in this paper is useful for comparing existing and future partitioning techniques. The best partitioning scheme may depend on the size of the problem, the computing system used, and the parallelism available.

In Fock matrix construction, each thread sums to its own copy of Fock submatrices in order to avoid contention for a single copy of the Fock matrix on a node. However, accelerators including Intel Xeon Phi have limited memory per core, making this strategy impossible for reduction across many threads. In effect, the problem size is limited when we run heterogeneously. Like many other applications, parallel HF calculations will benefit from accelerators that can directly access main DRAM memory on the node.

Finally, the main current challenge for improving Hartree–Fock performance is to speed up ERI calculations, which do not fully utilize SIMD capabilities on CPUs and Intel Xeon Phi. SIMD performance may be improved by grouping integrals of the same type, and computing them together using SIMD operations. This entails new interfaces between integral packages and the codes using them.

## Acknowledgments

The authors thank David Sherrill, Trent Parker, Rob Parrish, Aftab Patel, Yutong Lu, and the National Supercomputing Center in Guangzhou. This research was funded by the National Science Foundation under grant ACI-1147843 and an Intel Parallel Computing Center grant. Computer time for development on Stampede was provided under NSF XSEDE grant TG-CCR140016.

## References

- [1] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover, 1989.
- [2] 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*, vol. 181, no. 9, pp. 1477–1489, 2010.
- [3] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su *et al.*, “General atomic and molecular electronic structure system,” *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.
- [4] V. Lotrich, N. Flocke, M. Ponton, A. Yau, A. Perera, E. Deumens, and R. Bartlett, “Parallel implementation of electronic structure energy, gradient, and hessian calculations,” *The Journal of Chemical Physics*, vol. 128, p. 194104, 2008.
- [5] C. L. Janssen and I. M. Nielsen, *Parallel Computing in Quantum Chemistry*. CRC Press, 2008.
- [6] I. T. Foster, J. L. Tilson, A. F. Wagner, R. L. Shepard, R. J. Harrison, R. A. Kendall, and R. J. Littlefield, “Toward high-performance computational chemistry: I. Scalable Fock matrix construction algorithms,” *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 109–123, 1996.
- [7] R. J. Harrison, M. F. Guest, R. A. Kendall, D. E. Bernholdt, A. T. Wong, M. Stave, J. L. Anchell, A. C. Hess, R. J. Littlefield, G. I. 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*, vol. 17, pp. 124–132, 1996.
- [8] 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*, vol. 13, no. 4, pp. 291–302, 1999. [Online]. Available: <http://hpc.sagepub.com/content/13/4/291.abstract>
- [9] X. Liu, A. Patel, and E. Chow, “A new scalable parallel algorithm for Fock matrix construction,” in *2014 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Phoenix, AZ, 2014.
- [10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, “Scalable work stealing,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11. [Online]. Available: <http://doi.acm.org/10.1145/1654059.1654113>

- [11] A. Nikodem, A. V. Matveev, T. M. Soini, and N. Rösch, “Load balancing by work-stealing in quantum chemistry calculations: Application to hybrid density functional methods,” *International Journal of Quantum Chemistry*, vol. 114, no. 12, pp. 813–822, 2014. [Online]. Available: <http://dx.doi.org/10.1002/qua.24677>
- [12] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.
- [13] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, “Advances, applications and performance of the Global Arrays shared memory programming toolkit,” *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006. [Online]. Available: <http://hpc.sagepub.com/content/20/2/203.abstract>
- [14] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, “Supporting the Global Arrays PGAS model using MPI one-sided communication,” in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 739–750.
- [15] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, “Remote memory access programming in MPI-3,” Argonne National Laboratory, Preprint ANL/MCS-P4062-0413-1, 2013.
- [16] 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,” in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13) held as part of SC13*, 2013.
- [17] K. Yasuda, “Two-electron integral evaluation on the graphics processor unit,” *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2008.
- [18] I. S. Ufimtsev and T. J. Martinez, “Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation,” *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.
- [19] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, “Uncontracted Rys quadrature implementation of up to g functions on graphical processing units,” *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.
- [20] K. A. Wilkinson, P. Sherwood, M. F. Guest, and K. J. Naidoo, “Acceleration of the GAMESS-UK electronic structure package on graphical processing units,” *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2313–2318, 2011.
- [21] N. Luehr, I. S. Ufimtsev, and T. J. Martínez, “Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs),” *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011.
- [22] Y. Miao and K. M. Merz, “Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations,” *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, 2013.
- [23] T. Ramdas, G. K. Egan, D. Abramson, and K. K. Baldrige, “On ERI sorting for SIMD execution of large-scale Hartree-Fock SCF,” *Computer Physics Communications*, vol. 178, no. 11, pp. 817–834, 2008.

- [24] ———, “ERI sorting for emerging processor architectures,” *Computer Physics Communications*, vol. 180, no. 8, pp. 1221–1229, 2009.
- [25] N. Flocke and V. Lotrich, “Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations,” *Journal of Computational Chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.
- [26] M. Dupuis, J. Rys, and H. F. King, “Evaluation of molecular integrals over Gaussian basis functions,” *The Journal of Chemical Physics*, vol. 65, no. 1, pp. 111–116, 1976.
- [27] J. Rys, M. Dupuis, and H. F. King, “Computation of electron repulsion integrals using the Rys quadrature method,” *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 154–157, 1983.
- [28] R. McWeeny, “Some recent advances in density matrix theory,” *Rev. Mod. Phys.*, vol. 32, pp. 335–369, Apr 1960.
- [29] A. H. R. Palser and D. E. Manolopoulos, “Canonical purification of the density matrix in electronic-structure theory,” *Phys. Rev. B*, vol. 58, pp. 12 704–12 711, Nov 1998.
- [30] E. Dekel, D. Nassimi, and S. Sahni, “Parallel matrix and graph algorithms,” *SIAM Journal on Computing*, vol. 10, no. 4, pp. 657–675, 1981.
- [31] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM J. Res. Dev.*, vol. 39, pp. 575–582, 1995.
- [32] R. A. van de Geijn and J. Watts, “SUMMA: scalable universal matrix multiplication algorithm,” *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.
- [33] M. N. Ucisik, D. S. Dashti, J. C. Faver, and K. M. Merz, “Pairwise additivity of energy components in protein-ligand binding: The HIV II protease-indinavir case,” *J. Chem. Phys.*, vol. 135, p. 085101, 2011.
- [34] P. Pulay, “Convergence acceleration of iterative sequences. The case of SCF iteration,” *Chemical Physics Letters*, vol. 73, no. 2, pp. 393–398, 1980.
- [35] A. I. Krylov and P. M. Gill, “Q-Chem: an engine for innovation,” *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 3, no. 3, pp. 317–326, 2013. [Online]. Available: <http://dx.doi.org/10.1002/wcms.1122>