Parallel scalability of Hartree–Fock calculations

Edmond Chow,[1,a)] Xing Liu,[1] Mikhail Smelyanskiy,[2] and Jeff R. Hammond[2]
[1]*School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0765, USA*
[2]*Parallel Computing Lab, Intel Corporation, Santa Clara, California 95054-1549, USA*

(Received 28 November 2014; accepted 20 February 2015; published online 9 March 2015)

Quantum chemistry is increasingly performed using large cluster computers consisting of multiple interconnected nodes. For a fixed molecular problem, the efficiency of a calculation usually decreases as more nodes are used, due to the cost of communication between the nodes. This paper empirically investigates the parallel scalability of Hartree–Fock calculations. The construction of the Fock matrix and the density matrix calculation are analyzed separately. For the former, we use a parallelization of Fock matrix construction based on a static partitioning of work followed by a work stealing phase. For the latter, we use density matrix purification from the linear scaling methods literature, but without using sparsity. When using large numbers of nodes for moderately sized problems, density matrix computations are network-bandwidth bound, making purification methods potentially faster than eigendecomposition methods. © *2015 AIP Publishing LLC.* [http://dx.doi.org/10.1063/1.4913961]

I. INTRODUCTION

Quantum chemistry codes must make efficient use of parallel computing resources in order to reduce execution time. This is true for simulating both large and small molecular systems, as parallel hardware is now unavoidable. This paper studies the scalability of Hartree–Fock (HF) self-consistent field (SCF) iterations on distributed memory commodity clusters, i.e., computers consisting of multiple interconnected compute nodes. Scalability refers to the ability of an algorithm and/or its implementation to continue to reduce execution time on a fixed problem as the amount of parallel computing resources is increased. In practice, codes are not perfectly scalable due to the portion of execution time that is spent performing communication. As the number of nodes is increased, execution time may no longer decrease or may even increase if communication dominates the total time.

In this paper, we focus on the HF method and moderately sized problems, from about 100 to 1000 atoms. Larger problems are better handled by linear scaling methods.[1] We also limit the problem size because smaller problems are more challenging to parallelize efficiently and also give us a better idea of the impact of future computers with even more parallelism relative to the problem size. We note that at these problem sizes, the Fock and density matrices are treated as dense matrices, i.e., unlike in linear scaling methods, any sparsity is not exploited.

The HF method is a useful prototype for parallel scalability studies. Besides playing a fundamental role in electronic structure theory, being the starting point for most methods that treat electron correlation (both single- and multi-reference methods), it is very similar to hybrid density-functional theory (DFT), by virtue of inclusion of both Coulomb and exchange contributions; therefore, algorithmic and mathematical improvements in HF are readily extensible to some of the most popular methods in chemistry (e.g., B3LYP, among many other examples). Also, the computational components of HF ground-state energies contain the same bottlenecks as the evaluation of other molecular properties: atomic integral evaluation, contraction of atomic integrals with (density) matrices, and diagonalization. Further, many of the computational characteristics of the external exchange interaction that is the bottleneck in coupled-cluster singles and doubles (CCSD) resemble those of the Fock build that dominates the computation of the HF ground state energy in most cases.

Each HF-SCF iteration is composed of two major stages, which we analyze separately. The first stage is the computation of the Fock matrix, which involves the computation of electron repulsion integrals (ERIs) and combining these ERIs with elements of the density matrix. This stage is computationally very expensive due to the extremely large number of ERIs that must be computed. Although this stage is expected to be very scalable due to the large amounts of work that can be performed in parallel, recent research has shown that communication overhead in this stage can cause a significant decrease in speedup when large numbers of nodes are used.[2] Here, speedup refers to the factor by which a code is faster when multiple nodes (or processing units) are used, compared to using a single node, for a fixed problem.

The second stage in a HF-SCF iteration is the calculation of the density matrix. In HF calculations, this stage is traditionally performed by diagonalization, i.e., computing all the eigenvalues and eigenvectors of the Fock matrix. For the problem sizes we consider, the total amount of work in this stage is very small compared to the amount of work in the Fock matrix construction stage. However, diagonalization has much less parallelism than Fock matrix construction. Thus,

a)Electronic mail: echow@cc.gatech.edu

it is possible for the density matrix computation to limit performance on large numbers of nodes.

The contribution of this paper is to show empirically how Fock matrix construction and density matrix calculation affect the overall scalability of a HF-SCF algorithm. Since the relative scalability of these two components depends on problem size, we measure the performance of the components of an efficient implementation of HF-SCF for different problem sizes, and on different numbers of nodes. A significant amount of research has been dedicated to parallelizing Fock matrix construction (e.g., Refs. 3–11) and density matrix calculation (e.g., Refs. 12–14) but, to the best of our knowledge, the relative contribution of these two components to scalability and overall execution time of HF-SCF has not been studied.

In Sec. II, we describe the parallelization challenges of Fock matrix construction and specify an efficient implementation that we use for parallel scalability measurements. In Sec. III, we describe the use of purification for computing the density matrix. Developed in the $O(N)$ methods literature, purification uses sparsity to obtain linear scaling. In our work on HF for moderately sized problems, we treat the Fock and density matrices as dense. We show that even in this case, purification has performance advantages over diagonalization in the case of highly parallel computations. Results of parallel scalability studies are presented in Sec. IV. For a small problem (122 atoms) on large numbers of nodes, the execution time for density matrix calculation can exceed that for Fock matrix construction. For larger problems (up to 1205 atoms), the execution time for Fock matrix construction dominates. In a sense that will be made precise later, Fock matrix construction and density matrix calculation impact overall scalability approximately equally. Section V concludes this paper.

II. FOCK MATRIX COMPUTATION

Whether or not the Fock and density matrices should be replicated or distributed across nodes depends on the size of the matrices, the number of nodes, and the available memory per node. For large matrices, distributing the data may be necessary. Distribution of the data may also be more efficient than replication for computations with large numbers of nodes, in order to avoid needing to synchronize copies of the data across all the nodes. In this paper, we focus on the distributed case and assign a rectangular block of matrix elements to each node.

The Fock matrix, F, is computed as

$$F_{ij} = H_{ij}^{\text{core}} + \sum_{kl} D_{kl} \left(2(ij|kl) - (ik|jl)\right), \tag{1}$$

where H^{core} is the core-Hamiltonian, D is a density matrix, and $(ij|kl)$ denotes an element of the ERI tensor. The computation of the ERIs is distributed among the nodes. Once they are computed, the ERIs are combined with elements of the density matrix to form elements of the Fock matrix. For Gaussian atom-centered basis sets, which we assume in this paper, a *shell* is defined as the set of basis functions corresponding to orbitals for an atom with the same energy and angular momentum. For efficiency, ERIs are computed in batches called *shell quartets*, defined as

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

where M, N, P, and Q are called *shell indices*. It is possible to similarly define an *atom quartet*, indexed by four atom indices.

Shell quartets may be *screened*, i.e., its ERIs treated as zero, if

$$\sqrt{\sigma(M,N)\sigma(P,Q)} \leq \tau, \tag{2}$$

where

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

and τ is a screening threshold. The two-dimensional quantity σ can be precomputed and stored. This type of screening, often called *Schwarz screening*,[15] is essential for reducing the computational cost of HF but also forces the computational data access pattern to be irregular and the parallelization to be more complicated.

We can now write the generic algorithm for distributed Fock matrix construction, shown as Algorithm I. The algorithm is based on shell quartet computations, in order to efficiently exploit symmetries and screening of the ERI tensor. In the algorithm, quantities such as F_{MN} and D_{PQ} denote *submatrices* of the Fock matrix F and density matrix D, respectively. Each of these submatrices reside on one of the nodes according to the partitioning of F and D.

There are two basic options for distributed parallelization of this algorithm. The first option is to "statically" partition the set of shell quartets such that the computation load across the nodes is balanced and such that the communication of the D and F submatrices is minimized. The second option is to "dynamically" schedule tasks onto nodes, where a task is a subset of all the shell quartets. The tasks are defined such that there are many more tasks than nodes, so that when a node completes a task, it retrieves a new task from a global queue of tasks. This procedure is naturally load balanced as long as the granularity of the tasks is fine enough.

A good static partition is hard to achieve and thus many codes, including NWChem,[16] use dynamic scheduling. In NWChem, a task corresponds to the shell quartets in some number of atom quartets, such that each node (or process) will be assigned approximately a certain number of tasks. Shell quartets in the same atom quartet tend to

ALGORITHM I. Distributed Fock matrix construction. The input to the algorithm is a set of atoms and their positions, a basis set, and a density matrix, D. The output is the Fock matrix, F.

for unique shell quartets $(MN|PQ)$ **do**
  if $(MN|PQ)$ is not screened **then**
    Compute shell quartet $(MN|PQ)$
    Receive submatrices D_{MN}, D_{PQ}, D_{NP}, D_{MQ}, D_{NQ},D_{MP}
    Compute contributions to submatrices F_{MN}, F_{PQ}, F_{NP}, F_{MQ}, F_{NQ}, F_{MP} according to Eq. (1)
    Send submatrices of F to their owner nodes
  end if
end for

TABLE I. Effect of work stealing to balance load in Fock matrix construction on 225 nodes, for four molecular systems. The headings "w/steal" and "w/o steal" denote whether the static partitioning is used with or without work stealing, respectively.

	Load balance ratio		Time (s)	
Molecule	w/steal	w/o steal	w/steal	w/o steal
1hsg_28	1.049	1.489	0.533	0.727
1hsg_38	1.045	1.326	7.777	9.627
1hsg_45	1.036	1.259	22.037	26.487
1hsg_90	1.035	1.152	110.967	123.198

have the same requirements for D and F submatrices, and thus communication requirements can be reduced. In general, larger tasks mean that more submatrices of D and F can potentially be shared within a task, but smaller tasks are better for load balance. Smaller tasks, however, also introduce higher dynamic scheduling cost, especially if the scheduler is centralized on a single node.

Our parallelization approach is a hybrid of the first and second options.[2] It uses a static partitioning so that all the submatrices of D needed by a node can be prefetched before the computation (which requires internode communication); with dynamic scheduling, the same submatrices of D may be fetched repeatedly by the same node for different tasks. Similarly, each node only needs to send submatrices of F once to their owner nodes. Thus communication can be reduced by using a static partitioning. An issue, however, is that good load balance is difficult to achieve by static partitioning. We address this issue by combining the static partitioning with a type of dynamic scheduling called "work stealing."[2,17–19] When a node finishes all the work assigned to it by the static partitioning, it "steals" tasks from other nodes. The work stealing phase acts to polish the load balance.

Table I shows the effect of adding a work stealing stage to the static partitioning to improve the load balance. Four test problems are used, listed in order from small to large, and are described in Sec. IV. In the table, "w/steal" and "w/o steal" denote whether the static partitioning is used with or without work stealing, respectively. The load balance ratio is the ratio of the maximum compute time to the average compute time for the ERI calculations and local updates of the Fock matrix over all nodes. The timings shown in the table are the overall time required for Fock matrix construction. The results show that, without stealing, the load balance ratio is worse for smaller problems, which is expected, and that work stealing greatly improves the load balance ratio. The timings also indicate that the relative impact of work stealing is greater for the smaller problems.

III. DENSITY MATRIX COMPUTATION

A. Purification

The density matrix in the HF-SCF method is

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

where C_{occ} is the matrix formed by the lowest energy eigenvectors of the Fock matrix corresponding to occupied orbitals, or those eigenvectors corresponding to eigenvalues smaller than the chemical potential. The density matrix is therefore a "spectral projector" of the Fock matrix, F. Given the eigendecomposition $F = U\Lambda_F U^T$, where U is the matrix of eigenvectors and Λ_F is the diagonal matrix of eigenvalues, the density matrix is $D = U\Lambda_D U^T$ where the eigenvalues shown in Λ_D are 1 for the occupied orbitals (or eigenvalues of Λ_F less than the chemical potential) and 0 otherwise. To find C_{occ}, a common method is to compute all the eigenvalues and eigenvectors of F. Note that the Fock and density matrices referred to in this section are in an orthogonalizing transformation basis.

Although this common method works well for small numbers of nodes, its performance is poor for large numbers of nodes, due to limited parallelism in the eigendecomposition. In the divide-and-conquer algorithm for computing the eigendecomposition, which is known to be preferable over the QR algorithm for large problems when eigenvectors are desired, complicated tree-like data structures are used in the parallelization.[20] Instead of speeding up, the code may "slow down" when the number of nodes increases beyond a point. In these cases, to avoid slowing down, it is advantageous to map the eigendecomposition to a smaller subset of nodes. However, the scalability still suffers because many nodes would be idle.

An alternative to eigendecomposition is to use any of a large number of "diagonalization-free" techniques that have been developed for *linear scaling* electronic structure methods; for a recent review, see Ref. 1. These methods avoid solving an eigenvalue problem and compute D directly from F. Computation of the density matrix can be accomplished in linear time for electronic systems with "nearsightedness" which translates to being able to approximate F and D by sparse matrices.

In this paper, we focus on density matrix purification techniques (see, e.g., Ref. 21 and the references therein) for computing the density matrix. Originally developed for linear scaling methods and used in conjunction with matrix sparsity, we advocate using this class of techniques also in the context of moderately sized HF-SCF problems without sparsity, for the high parallelism case. Although scaling with problem size remains cubic, scaling with node count can be much better than diagonalization techniques due to better parallel properties.

The most basic density matrix purification technique is McWeeny purification.[22] Starting with an appropriate initial guess D_0, McWeeny purification computes the iterates

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

until it is determined that the iterates have converged. As is evident, the algorithm is based on matrix multiplication and thus has much more parallelism and is easier to parallelize than methods based on eigendecomposition. It is thus potentially useful as an alternative to eigendecomposition when a large number of nodes are used.

Assuming that the eigenvalues of D_0 are between 0 and 1, McWeeny purification can be regarded as a fixed-point iteration that maps the eigenvalues of D_k less than 0.5 toward 0, and the eigenvalues greater than 0.5 toward 1. Thus D_0 must be a suitably scaled and shifted version of F such that its eigenvalues lie between 0 and 1, and the chemical potential

ALGORITHM II. Canonical purification.

Set D_0 using Eq. (3)
for k = 0, 1, ... until convergence **do**
  $c_k = \text{trace}(D_k^2 - D_k^3)/\text{trace}(D_k - D_k^2)$
  if $c_k \le 1/2$ **then**
    $D_{k+1} = \left((1-2c_k)D_k + (1+c_k)D_k^2 - D_k^3\right)/(1-c_k)$
  else
    $D_{k+1} = \left((1+c_k)D_k^2 - D_k^3\right)/c_k$
  end if
end for

is mapped to 0.5. To produce D_0, one requires estimating the extremal eigenvalues of F as well as knowing the chemical potential.

We use an extension of McWeeny purification that computes the density matrix knowing only the number of occupied orbitals, rather than the chemical potential. Several such extensions exist, the first being canonical purification.[23] Here, the trace of the iterates D_k, which corresponds to the number of occupied orbitals, is preserved from step to step. In trace-correcting purification,[24] the trace converges to the desired value, but it is allowed to change from step to step in order to accelerate convergence, especially in cases where the fraction of occupied orbitals is very low or very high. In trace-resetting purification[25] (see also Ref. 26), the trace constraint is only enforced on certain steps. Convergence can also be accelerated by using nonmonotonic polynomial mappings, if the eigenvalues around the chemical potential are known or can be bounded.[27]

In this paper, we use canonical purification as described in Ref. 23 and shown in Algorithm II. Its main cost per iteration is two matrix multiplications, like plain McWeeny purification. We use the stopping criterion $\|D_k - D_k^2\|_F < 10^{-11}$. The initial iterate, D_0, must have the same eigenvectors as F, have its eigenvalues lie between 0 and 1, and have the required trace. This is accomplished by shifting and scaling F as

$$D_0 = \frac{\lambda}{n}(\bar{\mu}I - F) + \frac{n_e}{n}I, \tag{3}$$

where n is the number of basis functions, n_e is the number of occupied orbitals, and where

$$\lambda = \min\left\{\frac{n_e}{F_{\max} - \bar{\mu}}, \frac{n - n_e}{\bar{\mu} - F_{\min}}\right\}$$

and

$$\bar{\mu} = \frac{\text{tr}(F)}{n}.$$

We use Gershgorin's theorem[28] to cheaply provide outer bounds $F_{\min}$ and $F_{\max}$ on the smallest and largest eigenvalues of F, respectively. The Lanczos algorithm[29] can alternatively be used to estimate the extremal eigenvalues.

Results of tests comparing the distributed parallel performance of canonical purification to that of eigendecomposition will be shown in Sec. IV. We note that trace-correcting purification may have lower computational cost than canonical purification, particularly for very low or very high partial occupancies.[24,30]

B. Parallel matrix multiplication

The purification algorithm spends most of its execution time performing two matrix multiplications, computing the square and cube of D_k. Many algorithms exist for distributed parallel matrix multiplication, and most can be categorized as 2D algorithms[31–34] or 3D algorithms,[32,35–37] depending on whether the data are distributed on a 2D or 3D mesh of nodes. In 3D algorithms, communication costs are reduced relative to 2D algorithms by replicating the input matrices $p^{1/3}$ times over the entire machine, where p is the number of nodes. Recently, "2.5D" matrix multiplication algorithms have been proposed,[38] to balance the costs of storage and communication.

We implemented a 2D algorithm called SUMMA (Scalable Universal Matrix Multiply),[34] which is also implemented as the PDGEMM function in ScaLAPACK. We also implemented a 3D algorithm, following Ref. 37. We refer to these as the 2D and 3D algorithms in the remainder of this paper. These algorithms were implemented so that we could separately measure the time for computation and communication. We have verified that the timings for our 2D algorithm are very similar to the timings for PDGEMM. Note that matrix symmetry is very difficult to exploit efficiently in distributed dense matrix multiplication; we found that the PDSYMM function in ScaLAPACK (which allows one matrix in a matrix multiplication to be symmetric) generally performed worse than PDGEMM. We have not attempted to exploit symmetry in our implementations, however, any efficient matrix multiplication code for symmetric matrices could be applied and would benefit density matrix purification.

Finally, we note that the Fock matrix, which is scaled and shifted to form D_0, is initially partitioned in 2D fashion (see Sec. II). Thus, there is an additional communication cost in the 3D case over the 2D case to map D_0 into the required 3D data distribution. This cost, however, can be amortized over the many matrix multiplies that are used in the purification procedure.

Figure 1 compares the execution time for our 2D and 3D parallel matrix multiplication algorithms. We used a square number of nodes for the 2D algorithm and a cubic number of nodes for the 3D algorithm. Within each node, a multithreaded dgemm function (performing dense matrix multiplication from optimized linear algebra libraries) was called to perform the local matrix multiplications. We observe that the timings for the 2D and 3D algorithms are almost identical for small numbers of nodes. However, for large numbers of nodes (50 or more), the 3D algorithm is faster and appears to continue to scale well to the maximum number of nodes tested. The portion of the timings for the two algorithms spent in the dgemm function is also shown. For the case of the larger problem size in Figure 1(b), the timings for the dgemm function for the 2D and 3D algorithms are very similar for all numbers of nodes. The timings decrease perfectly linearly with increasing numbers of nodes. The discrepancy from the total 2D or 3D timings represents the communication time required by the algorithms. As shown, the 3D algorithm requires less communication than the 2D algorithm, and the difference between the two algorithms grows with increasing numbers of nodes.

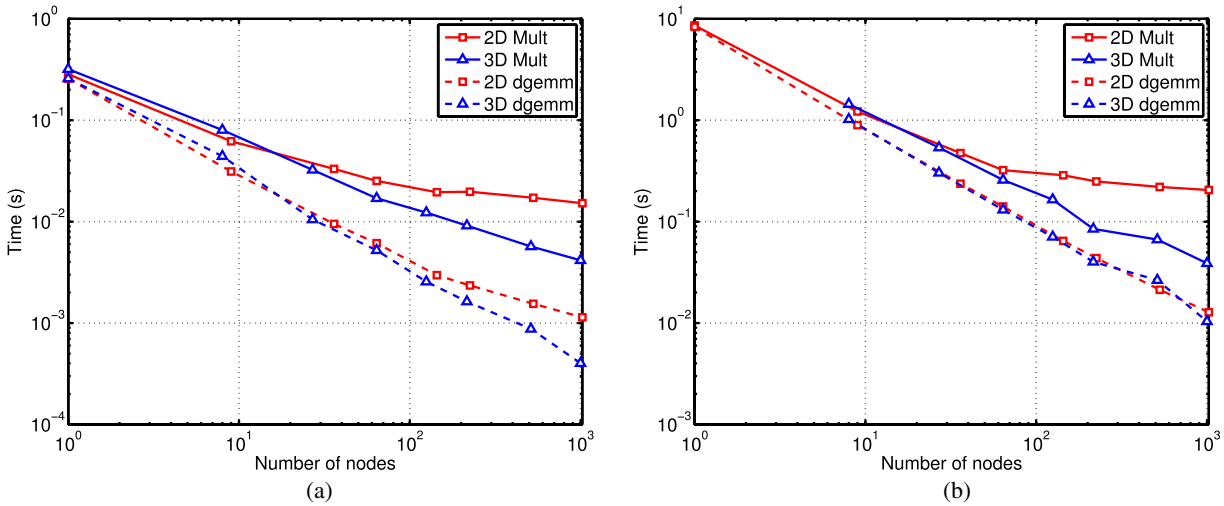


FIG. 1. Comparison of 2D and 3D matrix multiplication execution time vs. number of nodes for two molecular problem sizes. "dgemm" refers to time spent multiplying submatrices; the difference between this and the overall multiplication time is due to communication. (a) 3555 basis functions. (b) 11 163 basis functions.

For a fixed number of processors p and a fixed dimension n of the matrices, the matrix blocks have dimension $n/p^{1/2}$ in the 2D case and $n/p^{1/3}$ in the 3D case. Note that the blocks are larger in the 3D case. This means that the local matrix multiplications may be more efficient (up to a certain size depending on the hardware) in the 3D case because these multiplications involve larger submatrices. This effect can be observed for the smaller problem in Figure 1(a). Here, the dgemm timings are similar in both 2D and 3D algorithms for small numbers of nodes but are lower for the 3D case for larger numbers of nodes. This is due to lower efficiency of the dgemm function for smaller sizes.

Note that communication requires a large majority of the execution time on large numbers of nodes. In these cases, faster dgemm operations would not significantly improve the overall performance. Due to better performance of the 3D algorithm, we use the 3D algorithm for purification in the remainder of this paper.

IV. COMPUTATIONAL SCALING RESULTS

In this section, we first demonstrate the performance of the optimized implementations for Fock matrix construction and density matrix purification described in Secs. II and III. We refer to this code as GTFock. We then use GTFock to understand the relative importance of the scalability of these two components to the overall scalability of HF-SCF.

Tests were performed using 1 to 1024 nodes (16 to 16 384 cores) on the Stampede supercomputer located at Texas Advanced Computing Center. Each node is composed of two Intel Xeon E5-2680 processors (8 cores each at 2.7 GHz). Memory on these nodes is 32 GB DRAM. GTFock is coded in the C programming language and uses ScaLAPACK, MPI, and Global Arrays. We compiled GTFock using icc v14.0.1 and linked to Intel MKL v11.1 (for ScaLAPACK) and MVAPICH2 v2.0b (for MPI). Global Arrays uses ARMCI over InfiniBand on the Stampede machine.

Four molecular systems of different sizes were used to test scalability. The molecular systems were derived from a model of human immunodeficiency virus (HIV) II protease complexed with a ligand (indinavir). Atomic coordinates of all non-hydrogen atoms were obtained from the 1HSG crystal structure, neglecting H_2O molecules except for one closely bound H_2O found in between the protein and the ligand. Hydrogen atom coordinates were obtained via the H++ macromolecular protonation server.[39] We generated a set of test systems (Table II) by only including residues with any atom within a certain distance from any atom in the ligand. For a system named 1hsg_28, the distance is 2.8 Å. Peptide bonds cut during this procedure were capped by a hydrogen placed in the vector of the N-C peptide bond with a bond distance of 1.02 Å for N-H bonds and 1.10 Å for C-H bonds. All test molecular systems used the cc-pVDZ basis set.[40] A screening tolerance of $\tau = 10^{-10}$ was used for Schwarz screening of ERIs; see Eq. (2).

TABLE II. Test molecules, all with net charge of zero. The number of occupied orbitals is denoted by n_e. The band gap and HOMO/LUMO energies are in units of au. The eigenvalue spectrum for 1hsg_28 ranges from −20.6765 to 4.2688.

Molecule	Atoms	Shells	Functions	n_e	Band gap	HOMO	LUMO
1hsg_28	122	549	1 159	227	0.403 3	−0.298 6	0.104 7
1hsg_38	387	1 701	3 555	691	0.378 8	−0.298 1	0.080 7
1hsg_45	554	2 427	5 065	981	0.374 4	−0.297 6	0.076 8
1hsg_90	1 205	5 329	11 163	2 185	0.387 5	−0.301 7	0.085 8

HF calculations were performed for the spin-restricted case (RHF). For the SCF iterations, an initial guess for the density matrix was constructed using superposition of atomic densities (SAD). The iterations were accelerated by using direct inversion of the iterative subspace (DIIS).[41] For the four molecular systems, between 16 and 18 SCF iterations were required for convergence. We note that the systems are insulators and thus the convergence of purification, which depends on the HOMO-LUMO gap, is rapid. For the first density matrix calculation, canonical purification required between 34 and 36 iterations, and for the last density matrix calculation, purification required between 30 and 32 iterations for convergence for the four problems. Convergence will be much slower for small gapped systems, but the parallel scalability of purification remains the same. For a discussion of the convergence of purification with respect to the HOMO-LUMO gap, see, e.g., Ref. 30.

To give an idea of the total time required for the SCF procedure, 1hsg_28 required 204.7 s on 9 nodes of Stampede and 1hsg_90 required 956.9 s on 529 nodes of Stampede. These timings include the one-time cost of computing the canonical orthogonalization[42] transformation (2.3 and 12.2 s, respectively). We computed this transformation via an eigendecomposition. Alternatively, techniques from the linear scaling literature may be applied to compute an orthogonalizing transformation for very large molecular systems.[1]

A. Comparison to NWChem

We first compare the performance of GTFock to the performance of NWChem.[16] Figure 2 shows timings for Fock matrix construction and density matrix calculation for the two codes for the test system 1hsg_38. For GTFock, density matrix calculation used purification with the 3D matrix multiplication algorithm. For NWChem, density matrix calculation used the QR algorithm for eigendecomposition as implemented in the pdsyev function in ScaLAPACK.

For NWChem, the results show that Fock matrix construction scales up to about 144 nodes, but execution time increases with more nodes. Eigendecomposition, which requires only a small fraction of the execution time, scales poorly, and its execution time increases after 36 nodes. These results show that eigendecomposition never dominates the total time in NWChem for any number of nodes for this problem. Overall, the maximum speedup is 36 at 144 nodes. In general, better scalability would be observed for larger problems.

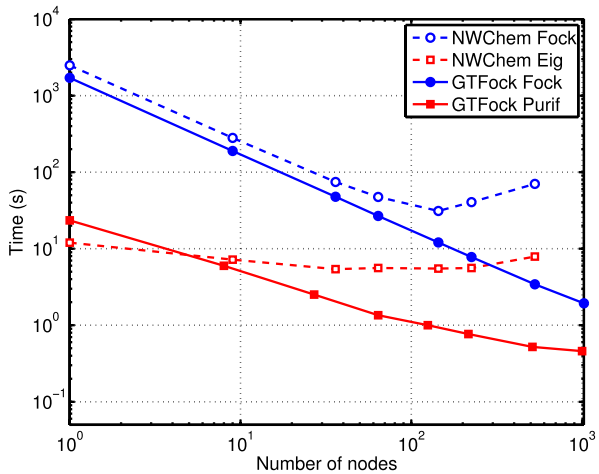


FIG. 2. Comparison of GTFock to NWChem for 1hsg_38 with 3555 basis functions.

In comparison, GTFock has better scalability than NWChem. Fock matrix construction scales up to 1024 nodes, which was the largest machine configuration we could test. Purification timings also decrease monotonically. (We note that in GTFock, Fock matrix construction used a square number of nodes, but purification was performed using a number of nodes that is the largest cube not exceeding that square.)

Fock matrix construction in NWChem is always at least a fixed factor slower than that in GTFock. This is because GTFock uses a slightly faster code for computing ERIs; this will be described in a future paper.

B. Different problem sizes

The overall scalability of HF-SCF is complex because it depends on two components, each with its own scalability characteristics, and the proportion of the computation spent on each component also changes in general with problem size. Understanding these issues helps code developers understand what are the bottlenecks for scalability. The results in Sec. IV B were for a single problem size, but we now analyze the timings for GTFock for different problem sizes.

The traditional analysis comparing the *problem size scalability* of Fock matrix construction and density matrix calculation argues that the latter dominates for large problems, rather than for small problems. This is because density matrix calculation (assuming dense matrices) scales as $O(n^3)$ arithmetic operations, while Fock matrix construction scales as $O(n^{2-3})$, e.g., see Ref. 43. However, in the case of very large numbers of nodes where calculations are network bandwidth bound (communication time dominates the computation time), problem size scaling does not depend on the cost of arithmetic operations, but rather on the cost of communication. For the same number of nodes, communication cost is relatively higher for smaller problems. Thus, in the case of large numbers of nodes leading to communication-bound performance, the execution time for density matrix calculation can dominate that for Fock matrix construction *for small problems rather than for large problems*.

The $O(n^3)$ scaling of density matrix calculation assumes compute-bound computation. Although the scaling of communication is typically less than $O(n^3)$, the absolute cost of communication is higher than the absolute cost of computation when the method is network bandwidth bound. For arbitrarily large problems, however, density matrix calculations will not be network bandwidth bound (given finite computer resources) and the calculations will scale as $O(n^3)$.

Figure 3 shows the scaling of execution time with the number of nodes for four problem sizes. The timings are separated into (1) Fock matrix construction, (2) density matrix calculation with eigendecomposition using the pdsyevd function[20] which implements the divide and conquer method

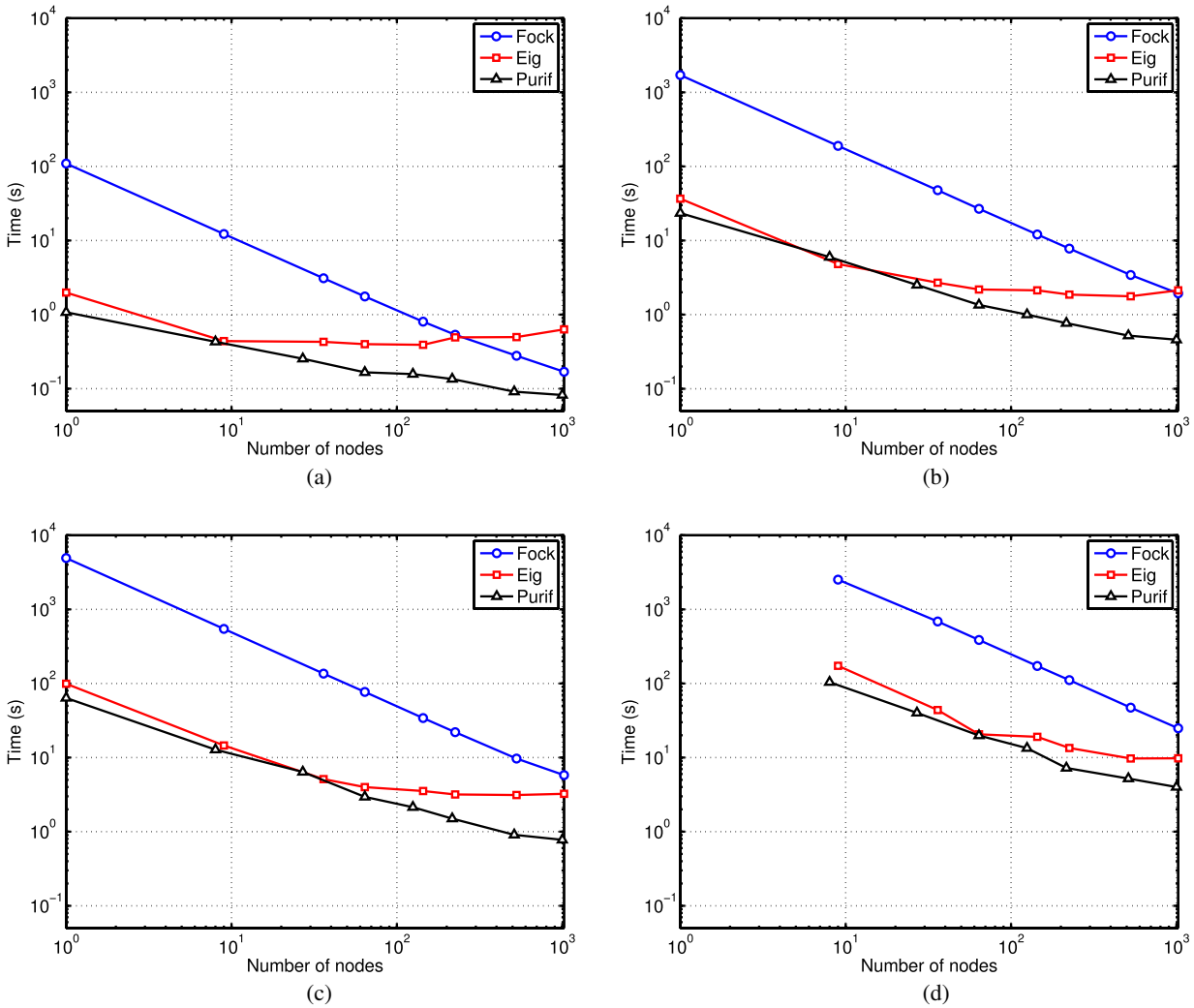


FIG. 3. Execution time vs. number of nodes for Fock matrix construction and density matrix computation for four molecular problem sizes. The largest problem could not be run on a single node due to memory limitations. (a) 1159 basis functions. (b) 3555 basis functions. (c) 5065 basis functions. (d) 11 163 basis functions.

in ScaLAPACK, and (3) density matrix calculation with canonical purification using Algorithm II.

As before, Fock matrix construction in GTFock shows good scaling for all problem sizes. Eigendecomposition and purification show poor scaling, with eigendecomposition scaling worse than purification. The eigendecomposition curve crosses the Fock matrix construction curve at 225 nodes for 1159 basis functions, and at 1024 nodes at 3555 basis functions. For larger problems, the intersection appears to be at a larger number of nodes. Thus we have the conclusion that the scalability of the eigendecomposition is more of a concern for small problems than for large problems. A similar conclusion can be drawn for purification, where it can be observed that the gap between Fock matrix time and purification time *grows* when going from 1159 to 5065 basis functions.

To analyze this further, note that the proportion of the time spent in eigendecomposition relative to Fock matrix construction is about 2 percent, with almost no growth for larger problem sizes, as measured by single node timings. For purification, this proportion is also almost constant, at 1 percent. For more nodes, these proportions are larger because of poorer scaling of density matrix calculations relative to Fock matrix construction. However, these calculations also scale better for larger matrices, which explains why density matrix calculation is more of a bottleneck for smaller problems rather than larger problems. The proportion of the time spent on density matrix calculation does not increase fast enough as problem sizes are increased for the bottleneck to appear at larger problem sizes. In general, *for the same number of nodes greater than 1,* as problem sizes are increased, the density matrix calculation time is smaller relative to Fock matrix construction time.

C. Strong scalability results

We now compare achieved performance to ideal performance. Figure 4 shows the speedup of Fock matrix construction and purification combined, as a function of the number of nodes. The actual speedup (Actual) improves for larger problem sizes, which is typical behavior, attaining approximately 80 percent efficiency for the largest problem size on 1024 nodes. What accounts more for this loss in parallel efficiency—Fock matrix construction or purification? Although the scalability of purification is poorer than for

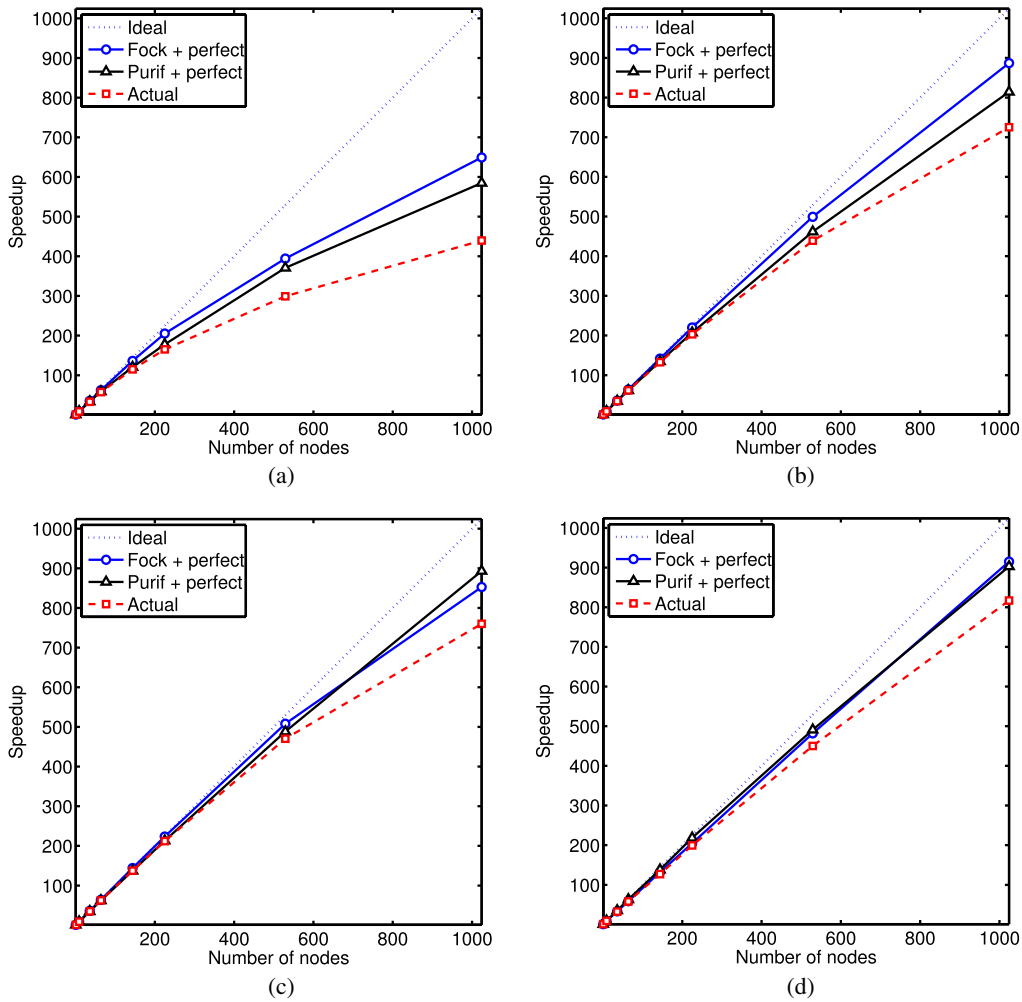


FIG. 4. Scalability of Fock matrix construction and purification combined, for four molecular problem sizes. Actual speedup is shown, along with projected speedup if purification is perfectly parallel (Fock + perfect) or if Fock matrix construction is perfectly parallel (Purif + perfect). In (d), scalability is relative to 9 nodes since this large problem could not be run on a single node. (a) 1159 basis functions. (b) 3555 basis functions. (c) 5065 basis functions. (d) 11 163 basis functions.

Fock matrix construction, the total time spent in purification is much less (see Figure 3). To answer the above question, Figure 4 also plots the total speedup using the actual Fock matrix construction timings but assuming purification is perfectly parallel (Fock + perfect), and total speedup using actual purification timings assuming Fock matrix construction is perfectly parallel (Purif + perfect). These two plots help identify the impact of each of the components on total

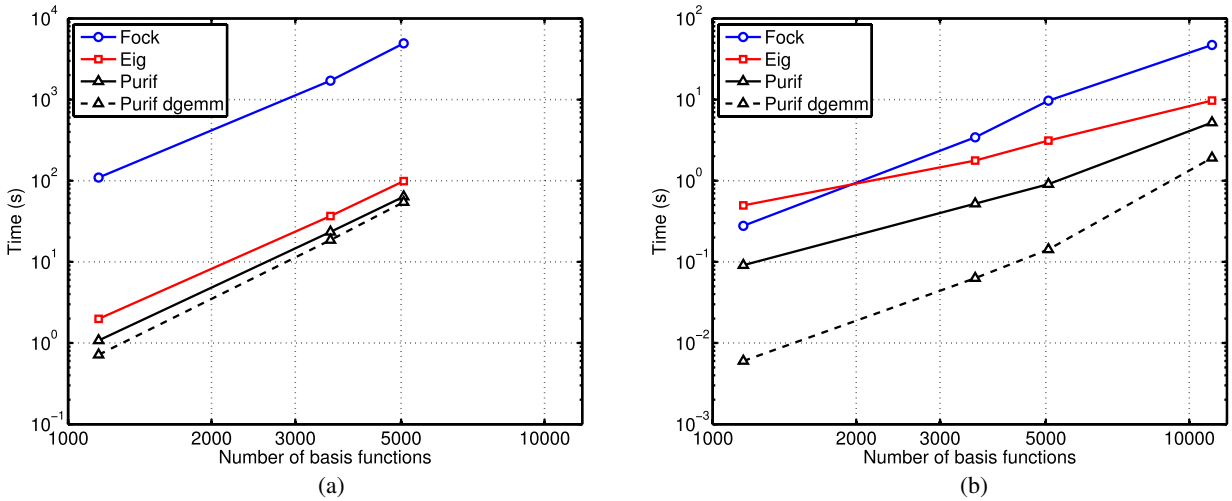


FIG. 5. Time vs. number of basis functions for computations using 1 and 529 nodes. (a) 1 node. (b) 529 nodes.

TABLE III. Scaling exponents with number of basis functions, computed using 1159 and 3555 basis functions.

	1 node	529 nodes
Fock	2.45	2.24
Eig	2.60	1.13
Purif	2.75	1.56
Purif dgemm	2.91	2.10

scalability. As can be seen, especially for the largest problem size, the impact on total scalability by the two components is about the same. Thus, one cannot say that scalability is impacted more by Fock matrix construction or by purification; both impact the overall scalability by about the same amount, due to the smaller amount of time spent in the less scalable density matrix calculation.

D. Scaling with number of basis functions

For n basis functions, the number of non-screened ERIs that must be computed is $O(n^{2-3})$, which is expected to be similar to the scaling of Fock matrix construction. Eigendecomposition using the divide and conquer algorithm nominally scales as $O(n^3)$ but may scale slightly better due to "deflation" in the algorithm. In purification, the matrix multiplication with dense matrices scales as $O(n^3)$.

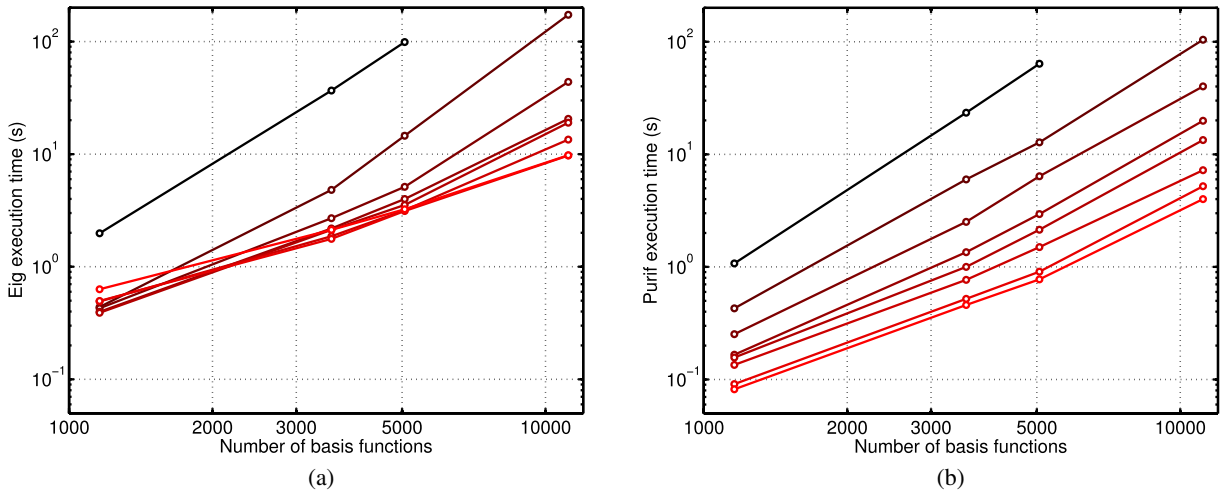
These values assume no cost for communication when multiple nodes are used. The communication cost will *reduce* the apparent scaling exponent because communication adds a large sub-$O(n^3)$ component to the overall computation cost. We illustrate this in several ways. Figure 5 plots the timings for various components of the computation as a function of number of basis functions for 1 node and 529 nodes. Included in this figure are the timings for the dgemm portion of the purification execution time, labelled "Purif dgemm." Table III shows the scaling exponents (slopes) for the curves shown in Figure 5, computed using 1159 and 3555 basis functions. For 1 node, eigendecomposition has a higher scaling exponent than Fock matrix construction. However, the reverse is true at 529 nodes. Both eigendecomposition and purification scaling exponents decrease significantly for 529 nodes, due to high communication cost relative to computation cost. We note also that the dgemm scaling exponent also degrades at 529 nodes; this is because of poorer efficiency of dgemm due to the use of small submatrices at this level of parallelism.

Figure 6 shows the scaling of eigendecomposition and purification with the number of basis functions. Each curve represents a different number of nodes. The slopes of the curves decrease with increasing numbers of nodes, corresponding to the degradation in scaling exponent due to communication. In particular, the slopes decrease faster in the eigendecomposition case compared to the purification case because execution time fails to decrease when increasing the number of nodes for the 1159 basis function case. We also observe an apparent increase in the scaling exponent (slope increases) for larger numbers of basis functions; this is because communication time is a smaller factor of the overall time for large problems.

E. Projected performance given faster ERI calculations

ERI calculations are the substantial portion of Fock matrix construction. This portion of the calculation can be accelerated in many ways, for example, by partially caching the expensive-to-compute integrals and by using hardware accelerators such as graphics processing units and Intel Xeon Phi. A change in the cost of ERI calculations changes the scalability of Fock matrix construction and the overall scalability of HF-SCF iterations. In Figure 7, we show the projected performance of Fock matrix construction if ERI calculations are accelerated by a factor of 10 for the 1hsg_38 test system. The projected timings were computed as follows. We assume that the Fock matrix construction timings T_N for node count N are composed of a perfectly scalable portion equal to T_1/N and a communication overhead h_N,

$$T_N = T_1/N + h_N, \quad h_1 = 0,$$



(a) (b)

FIG. 6. Comparison of eigendecomposition and purification execution time vs. number of basis functions for various numbers of compute nodes. For the eigendecomposition (a), the dark to light lines from top to bottom are for, respectively, 1, 9, 36, 64, 144, 225, 529, 1024 nodes. For purification (b), the dark to light lines from top to bottom are for, respectively, 1, 8, 27, 64, 125, 216, 512, 1000 nodes.

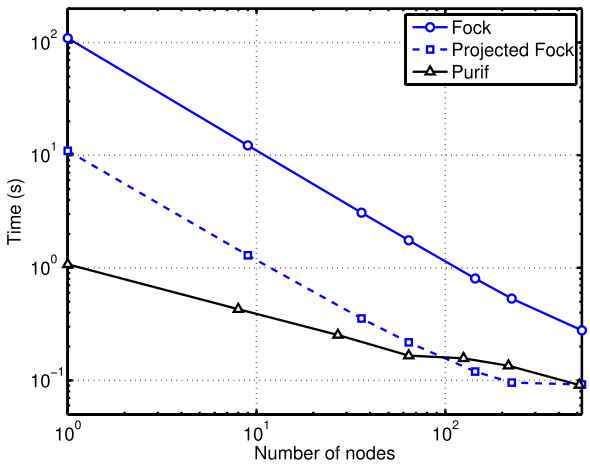


FIG. 7. Projected Fock matrix construction time ("Projected Fock") if ERI calculations are 10 times faster. The test system is 1hsg_28 with 1159 basis functions. Data for the solid blue and black curves are the same as that in Figure 3(a).

which gives a formula for computing h_N. The projected timings are then computed as

$$T_N^{\text{proj}} = T_1/(sN) + h_N,$$

where s is the acceleration factor, which is 10 in our case.

Figure 7 shows that the projected Fock matrix construction time ("Projected Fock") is smaller by a factor of 10 for small numbers of nodes, but at around 225 nodes, the communication overhead starts to dominate so that the time stops scaling. The projected Fock matrix construction time is small enough that the purification time starts to dominate at around 100 nodes, but Fock matrix construction time may start to dominate again for large node counts due to its lack of scalability. For larger matrices, the point at which the projected Fock timings flatten out would occur at a larger number of nodes. Similarly, loss of scalability would begin at a smaller number of nodes if the acceleration factor s is larger. The graph gives an interesting perspective of a potential future scenario where ERI calculations can be accelerated, but communication, whose costs are more difficult to reduce, stays the same.

V. CONCLUSIONS

Should code developers focus on optimizing Fock matrix construction because it requires the largest portion of the compute time, or should developers focus on the density matrix calculation because it scales poorly and may dominate the total time for large numbers of nodes? In this paper, we have addressed the parallel efficiency of both Fock matrix construction and density matrix calculations. The results show that there is not just one impediment to better scalability—both components of HF-SCF are important and almost equally impact overall scalability.

The more difficult challenge, however, lies in the efficient parallelization of density matrix calculations for *small* problems. (For large problems, a higher ratio of computation to communication improves scalability.) We have suggested using density matrix purification techniques as a potentially more scalable approach than eigendecomposition approaches for HF-SCF. Purification is already well established for linear scaling methods, but its applicability to highly parallel HF computations does not seem to be appreciated. Purification with dense matrices will require more arithmetic operations than eigendecomposition, but on modern computer architectures, data movement and parallelism are more important.

Although we expect the general trends shown in this paper to hold, specific conclusions will differ for different computers and as algorithm and implementation improvements are made to codes such as parallel eigendecomposition. In particular, recent eigensolvers such as ELPA[14] are demonstrating better performance than the pdsyevd ScaLAPACK routine we used for comparisons. ELPA also uses the divide and conquer algorithm like pdsyevd but can use a two-step procedure for tridiagonalization. Results from ELPA-2[14] (compare Figure 3(a) of the reference to Figure 3(c) of this paper) show comparable performance to purification for 1000 cores but fail to scale beyond that. ELPA is much faster, however, at low levels of parallelism, which can be expected.

Software based on the ideas of this paper has been released in open-source form as the GTFock framework for distributed Fock matrix computation. We are currently integrating this software into the Psi4 quantum chemistry package.[44]

Because of the similarity between Hartree-Fock theory and Kohn-Sham DFT, the work presented here can be readily extended to DFT. Moreover, additional quantum mechanical methods may be formulated in terms of (generalized) Coulomb and exchange matrices, meaning that GTFock may form the core of future massively parallel codes for methods including MP2 and various forms of coupled cluster theory, symmetry-adapted perturbation theory (SAPT), configuration interaction singles (CIS) and RPA for excited electronic states, coupled-perturbed Hartree-Fock or DFT for analytical energy gradients, and others.

ACKNOWLEDGMENTS

The authors thank David Sherrill, Trent Parker, Rob Parrish, and Aftab Patel for assistance on this research. The authors also thank the referees for a very thorough review of this paper, which led to a substantial improvement to its presentation. This research was supported by the National Science Foundation under Grant No. ACI-1147843 and by Intel Corporation under an Intel Parallel Computing Center grant. Computer time for development on Stampede was provided under NSF XSEDE Grant No. TG-CCR140016.

[1]D. R. Bowler and T. Miyazaki, Rep. Prog. Phys. **75**, 036503 (2012).
[2]X. Liu, A. Patel, and E. Chow, in *2014 IEEE International Parallel and Distributed Processing Symposium, Phoenix, AZ, 2014* (IEEE, 2014), pp. 902–914.
[3]T. R. Furlani and H. F. King, J. Comput. Chem. **16**, 91 (1995).
[4]I. T. Foster, J. L. Tilson, A. F. Wagner, R. L. Shepard, R. J. Harrison, R. A. Kendall, and R. J. Littlefield, J. Comput. Chem. **17**, 109 (1996).
[5]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. Neiplocha, G. Thomas, D. Elwood, J. Tilson, R. Shepard, A. Wagner, I. Foster, E. Lusk, and R. Stevens, J. Comput. Chem. **17**, 124 (1996).
[6]T. R. Furlani, J. Kong, and P. M. W. Gill, Comput. Phys. Commun. **128**, 170 (2000).

[7]Y. Alexeev, R. A. Kendall, and M. S. Gordon, Comput. Phys. Commun. **143**, 69 (2002).
[8]H. Takashima, S. Yamada, S. Obara, K. Kitamura, S. Inabata, N. Miyakawa, K. Tanabe, and U. Nagashima, J. Comput. Chem. **23**, 1337 (2002).
[9]C. L. Janssen and I. M. Nielsen, *Parallel Computing in Quantum Chemistry* (CRC Press, 2008).
[10]K. Ishimura, K. Kuramoto, Y. Ikuta, and S. Hyodo, J. Chem. Theor. Comput. **6**, 1075 (2010).
[11]H. Umeda, Y. Inadomi, T. Watanabe, T. Yagi, T. Ishimoto, T. Ikegami, H. Tadano, T. Sakurai, and U. Nagashima, J. Comput. Chem. **31**, 2381 (2010).
[12]R. J. Littlefield and K. J. Maschhoff, Theor. Chim. Acta **84**, 457 (1993).
[13]A. T. Wong and R. J. Harrison, J. Comput. Chem. **16**, 1291 (1995).
[14]A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer, J. Phys.: Condens. Matter **26**, 213201 (2014).
[15]M. Häser and R. Ahlrichs, J. Comput. Chem. **10**, 104 (1989).
[16]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, Comput. Phys. Commun. **181**, 1477 (2010).
[17]R. D. Blumofe and C. E. Leiserson, J. ACM **46**, 720 (1999).
[18]J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, *SC'09* (ACM, New York, NY, USA, 2009), pp. 53:1–53:11.
[19]A. Nikodem, A. V. Matveev, T. M. Soini, and N. Rösch, Int. J. Quantum Chem. **114**, 813 (2014).
[20]F. Tisseur and J. Dongarra, SIAM J. Sci. Comput. **20**, 2223 (1999).
[21]A. Niklasson, *Linear-Scaling Techniques in Computational Chemistry and Physics*, Challenges and Advances in Computational Chemistry and Physics, Vol. 13, edited by R. Zalesny, M. G. Papadopoulos, P. G. Mezey, and J. Leszczynski (Springer, Netherlands, 2011), pp. 439–473.
[22]R. McWeeny, Rev. Mod. Phys. **32**, 335 (1960).
[23]A. H. R. Palser and D. E. Manolopoulos, Phys. Rev. B **58**, 12704 (1998).
[24]A. M. N. Niklasson, Phys. Rev. B **66**, 155115 (2002).
[25]A. M. N. Niklasson, C. J. Tymczak, and M. Challacombe, J. Chem. Phys. **118**, 8611 (2003).
[26]D. K. Jordan and D. A. Mazziotti, J. Chem. Phys. **122**, 084114 (2005).
[27]E. H. Rubensson, J. Chem. Theor. Comput. **7**, 1233 (2011).
[28]R. A. Horn and C. R. Johnson, *Matrix Analysis*, 2nd ed. (Cambridge University Press, New York, 2013).
[29]G. H. Golub and C. F. V. Loan, *Matrix Computations*, 4th ed. (Johns Hopkins, 2013).
[30]E. Rudberg and E. H. Rubensson, J. Phys.: Condens. Matter **23**, 075502 (2011).
[31]L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. thesis (Montana State University, 1969).
[32]E. Dekel, D. Nassimi, and S. Sahni, SIAM J. Comput. **10**, 657 (1981).
[33]G. C. Fox, S. Otto, and A. J. G. Hey, Parallel Comput. **4**, 17 (1987).
[34]R. A. van de Geijn and J. Watts, Concurrency: Pract. Exper. **9**, 255 (1997).
[35]J. Berntsen, Parallel Comput. **12**, 335 (1989).
[36]A. Aggarwal, A. K. Chandra, and M. Snir, Theor. Comput. Sci. **71**, 3 (1990).
[37]R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, IBM J. Res. Dev. **39**, 575 (1995).
[38]E. Solomonik and J. Demmel, *Euro-Par 2011 Parallel Processing*, Lecture Notes in Computer Science Vol. 6853, edited by E. Jeannot, R. Namyst, and J. Roman (Springer, Berlin, Heidelberg, 2011), pp. 90–109.
[39]R. Anadakrishnan, B. Aguilar, and A. V. Onufriev, Nucleic Acids Res. **40**, W537 (2012).
[40]T. H. Dunning, Jr., J. Chem. Phys. **90**, 1007 (1989).
[41]P. Pulay, Chem. Phys. Lett. **73**, 393 (1980).
[42]A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory* (Dover, 1989).
[43]T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. Willems, Parallel Comput. **37**, 783 (2011).
[44]J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, and T. D. Crawford, WIREs Comput. Mol. Sci. **2**, 556 (2012).