Techniques for High-Performance Construction of Fock Matrices

H. Huang,[1] C. David Sherrill,[2] and E. Chow[1, a)]

[1)] *School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, USA*

[2)] *School of Chemistry and Biochemistry and School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, USA*

(Dated: 13 January 2020)

This paper presents techniques for Fock matrix construction that are designed for high performance on shared and distributed memory parallel computers when using Gaussian basis sets. Four main techniques are considered. (1) To calculate electron repulsion integrals, we demonstrate batching together the calculation of multiple shell quartets of the same angular momentum class so that the calculation of large sets of primitive integrals can be efficiently vectorized. (2) For multithreaded summation of entries into the Fock matrix, we investigate using a combination of atomic operations and thread-local copies of the Fock matrix. (3) For distributed memory parallel computers, we present a globally-accessible matrix class for accessing distributed Fock and density matrices. The new matrix class introduces a batched mode for remote memory access that can reduce synchronization cost. (4) For density fitting, we exploit both symmetry (of the Coulomb and exchange matrices) and sparsity (of 3-index tensors) and give a performance comparison of density fitting and the conventional direct calculation approach. The techniques are implemented in an open-source software library called GTFock.

Keywords: Coulomb and exchange matrix construction, parallel computing, global arrays, vectorization, electron repulsion integral, density fitting

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

I. INTRODUCTION

The complexity of modern computer hardware makes it challenging to implement computational chemistry programs that substantially exploit the performance capabilities of such hardware. This paper presents efficient techniques for constructing Fock matrices on shared and distributed memory computers when using Gaussian basis sets. The Coulomb and exchange components of the Fock matrix are often computed separately, as they are needed separately in different methods, but we will usually simply refer to this as constructing a Fock matrix. Let M, N, P, Q denote shell indices. Then blocks of the Coulomb and exchange matrices are

$$J_{MN} = \sum_{PQ} (MN|PQ) D_{PQ} \quad \text{and} \quad K_{MN} = \sum_{PQ} (MP|NQ) D_{PQ},$$

respectively, where D is the density matrix and $(MN|PQ)$ denotes a shell quartet of electron repulsion integrals (ERI). The Coulomb and exchange matrices are not computed one block at a time. Instead, to take advantage of the 8-way permutational symmetry in the ERI tensor, one loops over the unique shell quartets that survive screening and computes contributions to Coulomb and exchange matrix blocks (see Algorithm 1).

There are several pressure points that limit the ability to attain high performance when constructing a Fock matrix. The first is the unstructured nature of ERI calculation, which does not naturally lend itself to the SIMD or vector-like operations available on mod-

Algorithm 1 Generic structure of 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:     $J_{MN}$ += $(MN|PQ)D_{PQ}$
5:     $J_{PQ}$ += $(PQ|MN)D_{MN}$
6:     $K_{MP}$ += $(MN|PQ)D_{NQ}$
7:     $K_{NP}$ += $(NM|PQ)D_{MQ}$
8:     $K_{MQ}$ += $(MN|QP)D_{NP}$
9:     $K_{NQ}$ += $(NM|QP)D_{MP}$
10:  end if
11: end for
```

ern CPUs. To exploit these operations, the calculation must be reorganized such that multiple primitive or contracted integrals of the same type are computed simultaneously. Some work on ERI vectorization has been done in the past in the era of pipelined vector supercomputers[1–3] and about a decade ago for graphics processing units (GPUs)[4–9]. Due to the large temporary memory requirement per integral for integrals involving high angular momentum functions, however, how to best compute such integrals on GPUs and other accelerators with limited memory remains an open problem.

Most recently developed integral packages have not focused on SIMD efficiency[10–12]. An exception is the Simint package[13] which generates code for Obara–Saika recurrence relations for different CPU microarchitectures. Simint vectorizes the calculation of multiple *primitive* integrals of the same class (involving functions with the same combination of four total angular momentum numbers). For lightly-contracted basis sets, where atomic basis functions are linear combinations of small numbers of Gaussian-type functions, there are only small numbers of primitive integrals in a contracted integral. In this case, it is necessary to batch together the calculation of primitive integrals from multiple *contracted* integrals of the same class[14]. We discuss this in more detail in Section 2.

A second pressure point is the memory contention that arises when multiple threads on a multicore processor attempt to accumulate the product of ERIs and density matrix elements to the same Fock matrix location. Typically, this cost is small compared to the cost of ERI calculation. However, in the case of lightly-contracted and moderately-contracted basis sets (such as cc-pVDZ), where contracted integrals are less expensive to compute than for highly contracted basis sets, the time required for an efficient vectorized ERI calculation can be smaller than the time required for multithreaded accumulation of the Fock matrix elements. Thus the performance of this step of Fock matrix construction must also be optimized and some research has recently specifically addressed this[14,15]. We discuss Fock matrix accumulation in more detail in Section 3. Optimizations that improve cache performance and reduce vectorization overhead are also discussed in that section.

A third pressure point is load imbalance and communication cost when the Fock matrix is computed on distributed memory parallel computers. When the number of compute nodes is not large, each node can store its own partial sum of the Fock matrix and its own copy of the density matrix. This is known as the replicated data approach. Global communication is needed to form the fully-summed Fock matrix and to broadcast the density matrices. When the number of nodes is large, global communication must be avoided and a distributed data approach is needed. In this case, the Fock and density matrices are partitioned into blocks and the blocks are stored among the nodes. Communication occurs when a process computing contributions to Fock matrix blocks on one node needs to read density matrix blocks stored on other nodes; communication also occurs when contributions to Fock matrix blocks are accumulated on the nodes that store those blocks.

A large number of techniques for the distributed data approach have been proposed, for example, Refs 15–27. The main distinction between these techniques is whether the workload is partitioned statically among the nodes or whether chunks of work are assigned dynamically. In the static case, the partitioning can be chosen to minimize communication but load balance is difficult to achieve. In the dynamic case, communication patterns are not predictable, but load balance can be achieved naturally since work is dynamically assigned to processes when they become idle. The size of each chunk of work must be chosen carefully; larger chunks can reduce the required communication volume, but chunks that are too large lead to imbalance. GTFock uses a distributed data approach on distributed memory computers. Static partitioning is used, with work stealing when processes become idle[25,26,28], combining some of the advantages static and dynamic techniques at the cost of additional complexity.

For the communication of the Fock and density matrix blocks in the distributed data approach, quantum chemistry packages that support large-scale distributed memory computation typically use a distributed matrix class to simplify access to the globally accessible but distributed Fock and density matrices. Global indices, or more generally, pointers in a global memory address space can be used to access data that is partitioned among multiple compute nodes. Such a style of programming is called a partitioned global address space (PGAS) model and there are several PGAS matrix libraries. The Global Arrays toolkit[29] is a PGAS matrix library used in NWChem[22], Molpro[30], and GAMESS-UK[31]. The PPIDDv2 library[32] used in Molpro calls either Global Arrays or Message Passing Interface (MPI) functions. The Generalized Distributed Data Interface (GDDI)[33] is a PGAS framework for one-sided distributed memory data access and is used in GAMESS[15]. DArray[34] is a distributed array library developed for a Hartree–Fock code based on the UPC++[35] PGAS extension of C++. All these libraries provide one-sided communication because, in distributed Fock matrix construction, the compute node containing data needed by a process on another node, for example, may not be aware that it needs to participate in the communication.

Previously, GTFock used Global Arrays for one-sided distributed matrix access, which in turn used other low-level communication libraries. Global Arrays was developed at a time when there was no portable library for one-sided communication. To improve the portability of GTFock, now that MPI-3 is widely available for such communication, we have developed a lightweight PGAS matrix library called "GTMatrix" that only depends on MPI-3. GTMatrix provides only fundamental functionality: distributed storage, one-sided communication access to a global matrix, and a distributed task counter. These are sufficient for distributed Fock matrix construction. Different from other PGAS matrix libraries, GTMatrix also provides a new batched mode of communication in order to try to improve performance of certain types of matrix accesses when the number of compute nodes is large. GTMatrix is presented in Section 4.

An algorithmic approach to reduce the cost of constructing Coulomb and exchange matrices is to use density fitting[36–39] or any of its many variants, see e.g., the references in Ref. 40. Density fitting is a main feature in some codes[41] and distributed memory parallel codes have also been developed[42–45]. In addition to the conventional direct approach, GTFock can construct Fock matrices using density fitting. In Section 5, we discuss exploiting symmetry (of the Coulomb and exchange matrices) and sparsity (of 3-index tensors) in density fitting.

In Section 6, the performance of GTFock is demonstrated. As direct approaches for constructing the Fock matrix have been sped up, the regime of molecular system sizes where density matrix approaches is superior may now be smaller. Section 6 also presents an up-to-date comparison of Fock matrix construction performance between direct and density fitting approaches for different configurations and sizes of molecular systems.

An open-source implementation of the techniques described in this paper is available in the GTFock library at `https://github.com/gtfock-chem`. GTFock can be used to aid adoption of these techniques in new and existing quantum chemistry programs. GTFock supports computing multiple Coulomb and exchange matrices from multiple density matrices with a single calculation of the shell quartets. Nonsymmetric density matrices are also supported, although GTFock implements some optimizations when the density matrices are symmetric.

II. BATCHING AND VECTORIZATION OF ERI CALCULATIONS

To calculate ERIs and certain one-electron integrals, GTFock uses the Simint package[13] which was designed to effectively use SIMD operations. Simint calculates the contracted integrals of a shell quartet by vectorizing the calculation of primitive integrals. The Simint interface is similar to that of other integral packages: given shell pairs (M, N) and (P, Q) and their associated data, Simint computes the integrals in the shell quartet $(MN|PQ)$. However, Simint can also compute integrals for multiple shell quartets of the same class (involving functions with the same combination of four total angular momentum numbers) in one call, which is called "batching." Given a bra side shell pair list (M_i, N_i), $1 \leq i \leq l$ and a ket side shell pair list (P_j, Q_j), $1 \leq j \leq r$, Simint efficiently computes $(M_iN_i|P_jQ_j)$, $1 \leq i \leq l$, $1 \leq j \leq r$ in one call if all shell quartets $(M_iN_i|P_jQ_j)$ have the same class. Batching increases the number primitive integrals that can be computed in vectorized fashion and thus improves SIMD efficiency for lightly- and moderately-contracted basis sets. To batch together multiple shell quartets, the calling program needs additional code to perform batching on-the-fly.

Algorithm 2 shows how multithreaded ERI batching is implemented in GTFock. The M, N loop (line 2) is parallelized with OpenMP multithreading. Using the P, Q loop (line 3) as the parallelized outer loop is also possible, but the vectorization speedup of Simint in this case is likely to be smaller compared to that of using the M, N loop as the outer loop. Each thread maintains private shell quartet queues to generate shell quartet lists dynamically. If a shell quartet survives the uniqueness (permutational symmetry) and Schwarz screening tests, then it is pushed onto a queue according to the angular momentum (AM) of the P and Q shells (lines 4–6). When a queue is full, Simint is called to compute the integrals for all the shell quartets in the queue and these integrals are immediately used to compute Fock matrix blocks. The queue is then reset to empty (line 7–10). When a thread has looped over all P, Q pairs for a given M, N pair, it calls Simint with all its non-empty shell quartet queues and then resets these queues to empty (line 13–16). This multithreaded ERI batching algorithm does not need any locking. For additional details, see Ref. 14 where ERI batching using Simint was first described.

Table I illustrates the effect of Simint vectorization and of batching for different basis sets. The basis sets aug-cc-pVTZ, cc-pVDZ, and ANO-DZ can be considered to be

Algorithm 2 Multithreaded ERI batching.

```
 1: Each thread initializes its private queues to empty
 2: for shell pairs $M, N$ in parallel do
 3:   for shell pairs $P, Q$ do
 4:     if $(MN|PQ)$ is unique and not screened then
 5:       $k$ = index for the AM pair for shell pair $(P, Q)$
 6:       Push indices of $(MN|PQ)$ to queue $q[k]$
 7:       if queue $q[k]$ is full then
 8:         Compute (with Simint) and consume shell quartets in queue $q[k]$
 9:         Reset queue $q[k]$ to empty
10:       end if
11:     end if
12:   end for
13:   for a non-empty queue $q[i]$ do
14:     Compute (with Simint) and consume shell quartets in queue $q[i]$
15:     Reset queue $q[i]$ to empty
16:   end for
17: end for
```

lightly-contracted, moderately-contracted, and heavily-contracted basis sets, respectively. The baseline scalar timings used Simint code that was generated without SIMD operations. The test molecule, called protein-28, is a 30-atom system derived by truncating the 1hsg system from the protein data bank. Shell quartet screening (tolerance 10^{-11}) and primitive integral screening (tolerance 10^{-14}) were used. The tests were run using all 48 cores on a two-socket compute node with Intel Xeon Platinum 8160 processors.

For the heavily-contracted basis set, Simint vectorization is very effective due to a large number of primitive integrals per contracted integral, resulting in a long inner loop that can be vectorized. For the lightly- and moderately-contracted basis sets, vectorized Simint is not effective unless batching is used, which effectively increases the length of the inner loops.

The vectorization efficiency of Simint is different for different basis sets. Simint implements the Obara–Saika recurrence relations[46,47] in three steps: (1) vertical recurrence relations (VRR), (2) contractions, (3) horizontal recurrence relations (HRR). The VRR are vectorized much more efficiently than the HRR due to the layout of the computed quantities in memory. Because contractions are performed after the VRR and before the HRR, highly-contracted basis sets result in much more VRR work than HRR work. This helps explain why highly-contracted basis sets have better speedup over the scalar case than moderately-

and lightly-contracted basis sets even with batching.

TABLE I. ERI calculation timings (in seconds) for the protein-28 molecular system.

Basis Set	Basis Functions	Scalar Simint w/o Batching	Vectorized Simint w/o Batching	Vectorized Simint w/ Batching
aug-cc-pVTZ	1230	53.41	47.08	15.67
cc-pVDZ	310	0.281	0.256	0.127
ANO-DZ	526	693.40	180.68	186.44

III. MULTITHREADED FOCK MATRIX ACCUMULATION

Once a shell quartet of ERIs has been computed, the ERIs are combined with density matrix elements and accumulated into the Fock matrix. Algorithm 3 shows the sequential Fock matrix accumulation procedure given a shell quartet $(MN|PQ)$ that has just been computed. In the algorithm, ERI denotes the 4-dimensional array that contains the ERIs in $(MN|PQ)$, with dimensions $dimM \times dimN \times dimP \times dimQ$. A four-fold loop iterates over each element in the shell quartet and computes contributions to the Coulomb and exchange matrices, J and K. In practice, this procedure must be run by several threads simultaneously, with each thread computing and accumulating contributions associated with a subset of all the shell quartets, resulting in one final copy of J and K.

There are several approaches for parallelizing this procedure. A simple approach is to use atomic operations in lines 12–14, 16–17, and 19 to accumulate values into memory shared by all the threads[48] (the variables t_{MP}, etc., are stored in registers local to a thread). However, atomic operations require more underlying processor instructions than their non-atomic counterparts. Further, atomic operations can be a serial bottleneck if there are a large number of threads competing to write to the same location at the same time. An alternative is for each thread to sum into its own private ("thread-local") copy of J and K (e.g., Ref. 15). These partial sums are then summed together at the end. These final summations can be performed using a reduction operation between all threads. The final summations can also be performed with atomic operations, which are far fewer than if private copies of J and K were not used.

The above two approaches can be viewed as extremes where one copy of J and K is shared between all threads and where all threads sum into their own private copies of J

Algorithm 3 Fock matrix accumulation into matrices J and K, given shell quartet $(MN|PQ)$ stored in array ERI with dimensions $dimM \times dimN \times dimP \times dimQ$.

```
for $iM = 1$ to $dimM$ do
  for $iN = 1$ to $dimN$ do
    $t_{MN} = 0$
    for $iP = 1$ to $dimP$ do
      $t_{MP} = 0$
      $t_{NP} = 0$
      for $iQ = 1$ to $dimQ$ do
        $I = ERI(iM, iN, iP, iQ)$
        $t_{MN}$ += $D_{PQ}(iP, iQ) \times I$
        $t_{MP}$ += $D_{NQ}(iN, iQ) \times I$
        $t_{NP}$ += $D_{MQ}(iM, iQ) \times I$
        $J_{PQ}(iP, iQ)$ += $D_{MN}(iM, iN) \times I$
        $K_{MQ}(iM, iQ)$ += $D_{NP}(iN, iP) \times I$
        $K_{NQ}(iN, iQ)$ += $D_{MP}(iM, iP) \times I$
      end for
      $K_{MP}(iM, iP)$ += $t_{MP}$
      $K_{NP}(iN, iP)$ += $t_{NP}$
    end for
    $J_{MN}(iM, iN)$ += $t_{MN}$
  end for
end for
```

and K. In the latter, storage costs and bandwidth pressure are high when there are many threads. To balance the cost of using atomics and the cost of storage, an earlier version of GTFock used a hybrid of these two approaches for many-core Intel Xeon Phi processors as follows. Sets of four hyperthreads on one core share one copy of J and K and sum into this copy using atomic operations. All the copies of J and K, which are partial sums, are then summed using a reduction operation to produce the fully-summed J and K[26]. In general, it is important to find a balance between atomic operations and the number of copies of J and K.

A. Multithreaded Fock Matrix Accumulation with Thread-Local Buffers

1. *Block buffer accumulation*

Another way of balancing between the use of atomics and copies of J and K is to only use thread-local copies of the *blocks* of J and K that are needed for one shell quartet $(MN|PQ)$, i.e., only use copies of J_{MN}, J_{PQ}, K_{MP}, K_{MQ}, K_{NP}, K_{NQ}. However, these copies of the blocks of J and K must be accumulated to the global J and K (using atomics) before proceeding to the next shell quartet, rather than accumulating only after all shell quartets have been processed when copies of the full J and K are used. Compared to using copies of the full J and K, this technique dramatically reduces the amount of storage when there are many threads, at the cost of using more atomic operations. The amount of storage needed for each thread is very small, typically fitting in a CPU's L1 data cache (fastest memory local to a core). Compared to using a single shared copy of J and K, this technique uses far fewer atomics for a very modest increase in use of memory.

An optimization on top of this technique is that if two consecutive shell quartets (for a thread) have two indices in common, e.g., M and N, then the corresponding thread-local block of J or K, e.g., J_{MN} does not need to be accumulated into the global J (using atomics) after the first shell quartet, but the contribution of the second shell quartet to J_{MN} can be accumulated to the same thread-local buffer. This further reduces the number of atomic operations.

This technique was proposed in Ref. 14 and is shown here as Algorithm 4, which can be called block buffer accumulation ("BlockBufAcc"). The buffers j_{MN} are thread-local buffers corresponding to the global Fock matrix blocks J_{MN}, etc. The algorithm assumes that the shell quartets are processed in an order such that the fourth index Q changes most rapidly, so buffers involving this index cannot be reused between shell quartets. Performance results reported in Ref. 14 show that this algorithm is 2 to 3 times faster than the original hybrid algorithm used in GTFock mentioned above[26] on Intel Xeon Phi Knights Landing many-core processors.

2. Strip buffer accumulation

We now demonstrate an improvement over BlockBufAcc. The thread-local buffers used in BlockBufAcc can be viewed as partial J and K copies. Using partial J and K copies of a larger size would allow us to further reduce the number of atomic operations. Note that blocks K_{MP}, and K_{MQ} share the same first shell index and are located in the block row or "strip" of K that can be denoted as $K_{M,:}$ (Matlab colon notation). Similarly, the blocks K_{NP} and K_{NQ} are located in $K_{N,:}$. Let MS denote a thread-local buffer corresponding to

Algorithm 4 Fock matrix accumulation for shell quartet $(MN|PQ)$ using block buffers.

```
1: Initialize thread-local buffers $k_{MQ}$, $k_{NQ}$, $j_{PQ}$ to 0
2: for $iM = 1$ to $dimM$ do
3:     for $iN = 1$ to $dimN$ do
4:         for $iP = 1$ to $dimP$ do
5:             for $iQ = 1$ to $dimQ$ do
6:                 $I = ERI(iM, iN, iP, iQ)$
7:                 $j_{MN}(iM, iN) \mathrel{+}= D_{PQ}(iP, iQ) \times I$
8:                 $k_{MP}(iM, iP) \mathrel{+}= D_{NQ}(iN, iQ) \times I$
9:                 $k_{NP}(iN, iP) \mathrel{+}= D_{MQ}(iM, iQ) \times I$
10:                $j_{PQ}(iP, iQ) \mathrel{+}= D_{MN}(iM, iN) \times I$
11:                $k_{MQ}(iM, iQ) \mathrel{+}= D_{NP}(iN, iP) \times I$
12:                $k_{NQ}(iN, iQ) \mathrel{+}= D_{MP}(iM, iP) \times I$
13:            end for
14:        end for
15:    end for
16: end for
17: Atomically update $K_{MQ} \mathrel{+}= k_{MQ}$
18: Atomically update $K_{NQ} \mathrel{+}= k_{NQ}$
19: Atomically update $J_{PQ} \mathrel{+}= j_{PQ}$
20: if $M$ or $P$ will change in the next shell quartet then
21:     Atomically update $K_{MP} \mathrel{+}= k_{MP}$ and set $k_{MP}$ to 0
22: end if
23: if $N$ or $P$ will change in the next shell quartet then
24:     Atomically update $K_{NP} \mathrel{+}= k_{NP}$ and set $k_{NP}$ to 0
25: end if
26: if $M$ or $N$ will change in the next shell quartet then
27:     Atomically update $J_{MN} \mathrel{+}= j_{MN}$ and set $j_{MN}$ to 0
28: end if
```

$K_{M,:}$ and let NS denote a thread local buffer corresponding to $K_{N,:}$. Assuming that the indices M and N change slowly compared to indices P and Q as we loop through the shell quartets (on a thread), then the buffer MS can be continuously used by a thread until index M changes and MS must be accumulated to the global K (similarly for NS). Thus these buffers can be used for local accumulation of contributions to K_{MP}, K_{MQ}, K_{NP}, and K_{NQ} on a thread for many different values of the indices P and Q. For contributions to J_{MN} and J_{PQ}, we use two thread-local buffers j_{MN} and j_{PQ} like in Algorithm 4 since we assume the indices P and Q change rapidly.

Algorithm 5 shows this technique of using local buffers that correspond to strips of K, which we call strip buffer accumulation ("StripBufAcc"). In the algorithm, we use a subscript

Algorithm 5 Fock matrix accumulation for shell quartet $(MN|PQ)$ using strip buffers.

```
1: Initialize thread-local buffer $j_{PQ}$ to 0
2: for $iM = 1$ to $dimM$ do
3:     for $iN = 1$ to $dimN$ do
4:         for $iP = 1$ to $dimP$ do
5:             for $iQ = 1$ to $dimQ$ do
6:                 $I = ERI(iM, iN, iP, iQ)$
7:                 $j_{MN}(iM, iN) += D_{PQ}(iP, iQ) \times I$
8:                 $j_{PQ}(iP, iQ) += D_{MN}(iM, iN) \times I$
9:                 $MS_P(iM, iP) += D_{NQ}(iN, iQ) \times I$
10:                $MS_Q(iM, iQ) += D_{NP}(iN, iP) \times I$
11:                $NS_P(iN, iP) += D_{MQ}(iM, iQ) \times I$
12:                $NS_Q(iN, iQ) += D_{MP}(iM, iP) \times I$
13:            end for
14:        end for
15:    end for
16: end for
17: if $M$ and $N$ will change in the next shell quartet then
18:     Atomically update $J_{MN} += j_{MN}$ and reset $j_{MN}$ to zero
19: end if
20: Atomically update $J_{PQ} += j_{PQ}$
21: if $M$ will change in the next shell quartet then
22:     Atomically update $K$ with $MS$ and reset $MS$ to zero
23: end if
24: if $N$ will change in the next shell quartet then
25:     Atomically update $K$ with $NS$ and reset $NS$ to zero
26: end if
```

to indicate the block in MS and NS, e.g., Q in MS_Q. The storage required for MS and NS is not too large. Assuming fewer than 10,000 basis functions and basis sets containing up to g functions, the combined storage for MS and NS is less than 2.4 MB. When working with a block such as MS_P, the working set is small enough to fit in L1 data cache, like in Algorithm 4. A performance comparison of StripBufAcc and BlockBufAcc will be given in Section VI.

Using strip buffers appears to be first proposed by Mironov et al.[15] However, instead of using atomic operations to update the Fock matrix, Mironov et al.[15] synchronize the threads before updating the global Fock matrix from the data in the strip buffers. Mironov et al. compared using strip buffers with thread synchronization against using an entire Fock matrix copy for each thread. Timing results showed that the thread synchronization cost is low for small numbers of threads, but high for larger numbers of threads: when using 64 threads on an Intel Xeon Phi 7210 processor, using strip buffers with thread synchronization is about 50% slower than using thread-private Fock matrix copies.

B. Improving Memory Locality

In modern processors, data is transferred between memory and cache in blocks of fixed size called a "cache line." The typical cache line size for CPUs is 64 bytes. Assuming double precision floating point numbers, fetching a 3-by-3 block (72 bytes) from a matrix requires transferring 3 to 6 cache lines (192 to 384 bytes) from memory to the cache. The number of transferred cache lines depends on the size of the matrix. Considering that the Fock and density matrices are larger than 100×100 in most cases, the memory locality of accessing blocks in Fock and density matrices is very poor.

To improve the memory locality of accessing blocks in Fock and density matrices, we use a block storage scheme for these matrices. The matrix elements in a block corresponding to a shell pair are stored contiguously and an indexing scheme gives the actual location in memory of each block from the shell indices. The density matrix is packed into this storage scheme for Fock matrix construction. The Fock matrix is constructed in this storage scheme and then unpacked into standard dense matrix storage format when the construction is completed.

C. Reducing SIMD Overhead

After reducing the use of atomic operations, the performance of Fock matrix accumulation is still bounded by two factors: (1) the flop-per-byte ratio of Fock matrix accumulation is low, and (2) the length of the innermost loop, $dimQ$, which corresponds to the number of basis functions in a shell, is usually very small, which leads to high vectorization overhead. To address this second issue, we created five specialized kernels for Fock matrix accumulation, corresponding to different allowed values of $dimQ$: 1, 3, 6, 10, 15 (assuming Cartesian basis functions). In these specialized kernels, the length of the innermost loop is known at compile time, and the compiler unrolls these loops to reduce vectorization overhead. For $dimQ \geq 21$, we use a general kernel where we require the compiler to vectorize the innermost loop.

IV. GTMatrix: DISTRIBUTED MATRIX CLASS FOR ONE-SIDED GLOBAL MATRIX ACCESS

A. GTMatrix Interface

GTFock uses GTMatrix to store the density and Fock matrices in distributed fashion on distributed memory parallel computers. As mentioned in the Introduction, GTMatrix is implemented using MPI-3. GTMatrix uses 2D partitioning to distribute a matrix among a set of MPI processes with a user-specified process grid and user-specified sizes of each matrix block. Figure 1 shows an example of a matrix distributed over a 3×2 process grid.

GTMatrix supports three types of access operations: fetching a block of a global matrix to a local matrix ("get"), overwriting a block of a global matrix using values in a local matrix ("put"), and accumulating values of a local matrix to a block of a global matrix ("acc").

GTMatrix provides three modes for accessing a global matrix: blocking access, nonblocking access, and batched access. The blocking access mode is used for completing an access as soon as possible. When a blocking access function returns, the access is completed. The nonblocking access mode is used for overlapping an access operation with local computation or other access operations. When a nonblocking access function returns, the access is posted but not completed. The calling program must explicitly call the provided "wait" function to wait for the completion of nonblocking accesses. The Global Arrays toolkit provides both blocking and nonblocking access modes and UPC++ uses nonblocking access mode by

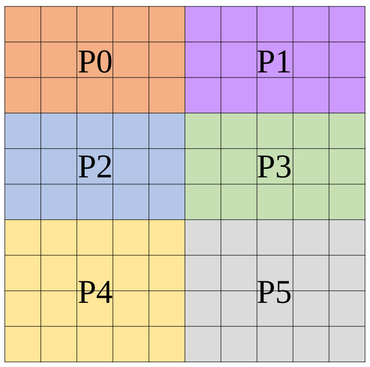

FIG. 1. GTMatrix of size 10×10 distributed over a 3×2 process grid. The partitioning of the matrix and the process grid are the same as those in Listing 2. The processes are numbered P0 to P5. Matrix entries that have the same color are stored in the same process, for example, all green matrix entries are stored on process P3. GTMatrix numbers the processes in row major order.

default for almost all operations.

Batched access mode is a new access mode introduced by GTMatrix. It is designed to accelerate two communication patterns: (1) a process making a large number of accesses to another process and (2) many-to-many communications. A batched access is performed in two stages: (1) submitting batched access requests and (2) completing the accesses. When a batched access request is submitted, it is queued instead of being posted immediately like in nonblocking access mode. Access operations are posted and completed when a program calls an "exec" function to explicitly complete these accesses. In this exec operation, a calling process only synchronizes with each target process once for all requests in order to reduce synchronization overhead, and a ring algorithm is used to accelerate many-to-many communications. This exec operation is blocking and cannot be overlapped with other operations. All local buffers associated with batched access requests cannot be reused until these batched access requests are completed.

Similar to Global Arrays, GTMatrix guarantees the atomicity of accumulating values into a global matrix in all access modes. For all put operations in batched mode, GTMatrix guarantees that all put operations are posted in the order of submission, but GTMatrix depends on the underlying MPI-3 implementation regarding the order of completion of these operations.

GTMatrix provides a C language interface with a syntax that is similar to the syntax of Global Arrays. Listing 1 shows this interface. All access functions use the same set of parameters as specified in lines 12–13. In this set of parameters, the block to be accessed is the `row_num`-by-`col_num` block whose top-left corner is (`row_start`, `col_start`) in the global matrix. The `src_buf` array is a local row-major buffer with leading dimension `src_buf_ld` that stores the data to be sent or the data after it is fetched.

Listing 2 gives some simple examples of the use of GTMatrix. Lines 3–11 create the GTMatrix shown earlier in Figure 1. All processes in the MPI communicator `mpi_comm` store one block of the global matrix as shown in the figure. Line 21 demonstrates the get operation using nonblocking access mode. Lines 25–29 demonstrate the accumulate operation using batched access mode. Note that get, put, and accumulate operations cannot be combined within a batched access epoch, as such combinations would depend on the order of the accesses, which would not be preserved when accesses are coalesced.

Listing 1. C interface of GTMatrix

```c
// Create and initialize a GTMatrix structure
int GTM_create(GTMatrix_t *gtm, MPI_Comm comm, MPI_Datatype datatype,
    int unit_size, int nrows, int ncols, int r_blocks, int c_blocks,
    int *r_displs, int *c_displs
);
// Destroy a GTMatrix structure
int GTM_destroy(GTMatrix_t gtm);

// Data access functions in GTMatrix have the same parameters,
// grouped in a macro here for convenience
#define GTM_PARAM \
    GTMatrix_t gtm, int row_start, int row_num, \
    int col_start, int col_num, void *src_buf, int src_buf_ld

// Blocking get / put / accumulate a block
int GTM_getBlock(GTM_PARAM);
int GTM_putBlock(GTM_PARAM);
int GTM_accBlock(GTM_PARAM);

// Nonblocking get / put / accumulate a block
int GTM_getBlockNB(GTM_PARAM);
int GTM_putBlockNB(GTM_PARAM);
int GTM_accBlockNB(GTM_PARAM);
// Wait for all outstanding nonblocking calls posted by
// the calling MPI process
int GTM_waitNB(GTMatrix_t gtm);

// Start a batched get / put / accumulation epoch and allow
// this process to submit get / put / accumulation requests
int GTM_startBatchGet(GTMatrix_t gtm);
int GTM_startBatchPut(GTMatrix_t gtm);
int GTM_startBatchAcc(GTMatrix_t gtm);

// Add a request to get / put / accumulate a block
int GTM_addGetBlockRequest(GTM_PARAM);
int GTM_addPutBlockRequest(GTM_PARAM);
int GTM_addAccBlockRequest(GTM_PARAM);

// Execute all get / put / accumulate requests in the queues
int GTM_execBatchGet(GTMatrix_t gtm);
int GTM_execBatchPut(GTMatrix_t gtm);
int GTM_execBatchAcc(GTMatrix_t gtm);

// Stop a batched get / put / accumulate epoch
int GTM_stopBatchGet(GTMatrix_t gtm);
int GTM_stopBatchPut(GTMatrix_t gtm);
int GTM_stopBatchAcc(GTMatrix_t gtm);

// Symmetrize a matrix, i.e. (A+A^T)/2, for int and double types
int GTM_symmetrize(GTMatrix_t gtm);

// Synchronize all processes
int GTM_sync(GTMatrix_t gtm);
```

Listing 2. Example Code Using GTMatrix

```
// Create a 10 * 10 matrix of type double
// distributed on a 3 row * 2 column process grid
GTMatrix_t gtm;
int nrow = 10, ncol = 10;
int n_rowblk = 3, n_colblk = 2;
int r_displs[4] = {0, 3, 6, 10};
int c_displs[3] = {0, 5, 10};
GTM_create(
    &gtm, mpi_comm, MPI_DOUBLE, sizeof(double),
    nrow, ncol, n_rowblk, n_colblk, &r_displs[0], &c_displs[0]
);

// After the GTMatrix is created, the local matrix block can be
// initialized by storing data in array gtm->mat_block in row-major
// order with leading dimension gtm->ld_local.
// GTM_sync(gtm) should be called after initialization
// to prevent premature access to the matrix.

// Use nonblocking mode to get a block [rs0:rs0+rn0-1, cs0:cs0+cn0-1] from
// global matrix to local row-major buffer buf0 with leading dimension ld0
GTM_getBlockNB(gtm, rs0, rn0, cs0, cn0, buf0, ld0);
GTM_waitNB(gtm);

// Use batched mode to accumulate multiple local blocks to global matrix
GTM_startBatchAcc(gtm);
for (int i = 0; i < num_acc_blk; i++)
  GTM_addAccBlockRequest(gtm,rs[i],rn[i],cs[i],cn[i],buf[i],ld[i]);
GTM_execBatchAcc(gtm);
GTM_stopBatchAcc(gtm);

// Destroy the GTMatrix structure
GTM_destroy(gtm);
```

B. Using GTMatrix Efficiently for Fock Matrix Construction

To reduce communication time, different GTMatrix access modes should be used for different access patterns. If a static shell quartet partitioning scheme is used, the necessary communication of the Fock and density matrix blocks is known before calculation starts. Since static partitionings are coarse (one partition for each node), each process usually needs to access a large number of blocks in the Fock and density matrices distributed on many different target processes. Batched access mode can be used to accelerate such many-to-many communications. Alternatively, each process can also partition all its shell quartets into multiple sub-tasks and use nonblocking access mode to overlap the accesses to the density matrix with local ERI calculations.

If a dynamic task scheduler is used, each process can partition its assigned shell quartets into multiple sub-tasks and use nonblocking access mode for fetching density matrix blocks such that communication is pipelined and overlapped with computation. If the workload in each sub-task is small, the extra cost of splitting and pipelining the communications may cancel out the performance improvement of overlapping communication with computation. The program may also use blocking access mode for fetching density matrix blocks before ERI calculation if the communication volume is relatively small. For accumulating the local Fock matrix sum to the global Fock matrix, the program could use batched access mode to accelerate the many-to-many communications.

We first describe the GTFock algorithm for distributed-memory Fock matrix construction before discussing how we use GTMatrix in GTFock. In GTFock, each process gathers blocks of the global density matrix it needs before commencing ERI calculations. We call this procedure "GatherD". After computing the contribution to Fock matrix blocks associated with all shell quartets in a task, each process does not immediately accumulate the contribution to Fock matrix blocks into corresponding global Fock matrix blocks. Instead, each process has a Fock matrix buffer containing Fock matrix elements that this process will access if no task stealing occurs. Contributions to the Fock matrix blocks of a task will be accumulated into the Fock matrix buffer on the process that this task originally belongs to. We call this procedure "AccFBuf". Compared to directly accumulating the contribution into corresponding global Fock matrix blocks, using a Fock matrix buffer as mentioned above has a smaller local memory footprint and smaller network communication volume. Finally, each process accumulates its Fock matrix buffer into corresponding global Fock matrix blocks. We call this procedure "ScatterF".

We use different access modes for different communication procedures in GTFock. For GatherD, we use batched access mode to accelerate many-to-many communication. For AccFBuf, we use nonblocking access mode since it is unnecessary to finish the accumulation immediately. For ScatterF, we use batched access mode to reduce synchronization cost and accelerate many-to-many communication since each process needs to update a large number of blocks on different target processes.

C. Implementation of GTMatrix

1. Passive target synchronization

Remote memory access operations in MPI-3 are nonblocking. Thus, a mechanism is needed to indicate when the access operations are completed. For example, we must know when a nonblocking get operation is completed so that we can use the result. GTMatrix uses MPI-3 passive target synchronization to signal completion, which only involves the origin process (i.e., caller of the get, put, or accumulate operation) and not the target process. The origin process calls an “unlock” function that waits or blocks until the access operation is completed.

2. MPI derived data types

MPI derived data types (DDTs) describe memory layouts and are convenient to use when data to be accessed is not laid out contiguously in memory. GTMatrix uses row-major storage for the portion of the matrix stored on a process. Accessing blocks containing just a portion of multiple rows stored on a process involves noncontiguous memory access. GTMatrix thus uses MPI DDTs, which can result in better performance than if rows of the block are sent with separate communication calls, or if packing and unpacking of the rows are performed explicitly by the sender and receiver. Packing and unpacking also interferes with the need to perform one-sided accesses. MPI DDTs specify how to pack and unpack data within the MPI library and can avoid unnecessary data copying on some hardware[49].

To reduce the overhead of creating and releasing MPI DDTs, GTMatrix predefines MPI DDTs for all possible sizes of matrix blocks up to 16×16 (this size can be adjusted at compile time). For accessing larger matrix blocks, a MPI DDT is defined and used just-in-time and released after posting the access operation.

We note that the blocked storage scheme in Section III B is used for GTFock’s internal representation of blocks of the density and Fock matrices (e.g., the Fock matrix buffer mentioned in the previous subsection), not for GTMatrix. A conversion between this internal representation and GTMatrix storage is needed, as mentioned in Section III B.

3. MPI shared memory

GTMatrix explicitly uses MPI shared memory to accelerate intra-node MPI process communication. Many MPI implementations can accelerate intra-node communication using shared-memory automatically, but manually and explicitly using shared memory for intra-node communication has a smaller overhead. When creating a global matrix, GTMatrix first creates a shared memory MPI window and allocates memory in this window. GTMatrix then creates a global MPI window using the memory allocated in the shared memory MPI window. For read operations, if the target process (the process containing the target data) are on the same node as the source process, GTMatrix uses direct memory copy in the shared memory MPI window instead of *MPI_Get*. For put and accumulation operations, GTMatrix always uses *MPI_Accumulate* in the global MPI window to guarantee the atomicity of these operations.

4. Nonblocking access mode

Nonblocking access mode in GTMatrix is different from nonblocking access mode in other PGAS frameworks. GTMatrix does not store or provide a handle for checking the status of each nonblocking access. Instead, GTMatrix only allows a program to wait for the completion of all outstanding nonblocking accesses. We made this design choice because the functions that provide a handle (such as *MPI_Rput* and *MPI_Raccumulate*, where "R" stands for "request handle") have larger overheads compared to those that don't (such as *MPI_Put* and *MPI_Accumulate*).

5. Batched access mode

Batched access mode in GTMatrix does not simply store all batched access requests and then perform access operations one by one. When a batched access request is submitted, this access request is decomposed into multiple single-target access requests such that each of these new requests has only one target process. Each single-target access request is then pushed onto a local queue associated with the target process. No actual communication operation is posted when a batched access request is submitted. All batched access requests are posted and completed when the application code calls GTMatrix explicitly to complete all access requests. The source process only synchronizes with each target process once for completing all access requests instead of synchronizing for each access request like in blocking access mode. A ring algorithm is used as the communication algorithm; a source process P_s completes its access requests to target process P_t in the following order: $t = s, s+1, \ldots, p-1, p, 1, 2, \ldots, s-2, s-1$. The ring algorithm aims to lower the likelihood of hotspots, i.e., data on one process being accessed by a large number of other processes at the same time.

V. DENSITY FITTING

Density fitting is a method of approximating the ERI tensor, for example,

$$(pq|rs) \approx \sum_{P,Q} (pq|P)[J^{-1}]_{PQ}(Q|rs)$$

where uppercase P and Q are indices of basis functions belonging to an auxiliary basis set and $[J]$ is the Coulomb fitting metric. In practice, we compute and store

$$B^Q_{pq} = \sum_P (pq|P)[J^{-1/2}]_{PQ} \tag{1}$$

so that

$$(pq|rs) \approx \sum_Q B^Q_{pq} B^Q_{rs}$$

although these integrals are never explicitly formed when computing the Coulomb and exchange matrices in density fitting.

Using B^Q_{pq} and the density matrix D, the Coulomb matrix J is computed in two steps,

$$V^Q = \sum_{r,s} B^Q_{rs} D_{rs} \tag{2}$$

$$J_{pq} = \sum_Q B^Q_{pq} V^Q. \tag{3}$$

In the usual case where J is symmetric, GTFock saves a factor of two in computations by computing V^Q above only for $r \geq s$ and only computing J_{pq} for $p \geq q$.

The exchange matrix K is also computed in two steps. Let C_{ri} denote the expansion coefficients for the i-th occupied molecular orbital. Then, assuming symmetry of the density matrix,

$$W_{pi}^Q = \sum_r C_{ri} B_{pr}^Q, \tag{4}$$

$$K_{pq} = \sum_Q \sum_i W_{pi}^Q W_{qi}^Q. \tag{5}$$

GTFock also exploits any symmetry in K as follows. We first note that (5) is a matrix multiplication and is most efficiently calculated using optimized library functions. GTFock partitions K into submatrices size $b \times b$ (nominally) and only computes the submatrices in the block upper triangular part of K using optimized matrix multiplication in the batched BLAS library, which supports simultaneous matrix multiplication of small submatrices[50]. If b is too small, the flop-per-byte ratio of matrix multiplication is not large enough to obtain good performance; larger b leads to more redundant computation in the diagonal submatrices. In practice, we choose $b = \max(64, n/10)$ for n basis functions.

Schwarz screening can be applied in density fitting to reduce the cost of certain computations[51]. From the upper bound provided by the Schwarz inequality, $(pq|P) \le \sqrt{(pq|pq)(P|P)}$, if $(pq|pq)$ is small then $(pq|P)$ can be neglected. More precisely, if σ is a threshold on the size of $(pq|P)$, then $(pq|P)$ can be neglected if

$$(pq|pq) < \frac{\sigma^2}{\max_P (P|P)}, \tag{6}$$

where $\max_P(P|P)$ is the maximum value of $(P|P)$ over all P. In practice, screening is performed per pair of shells and then, for those shell pairs that survive, screening is performed for basis function pairs within those shell pairs.

The sparsity pattern of B_{pq}^Q is the same as the sparsity pattern of $(pq|P)$ after screening since contracting $(pq|P)$ with the dense matrix $[J^{-1/2}]_{PQ}$ does not change the pq sparsity. With a sparse B_{pq}^Q, memory requirements are reduced and the cost of computing (2), (3), and (4) is reduced. However, (5) does not benefit from screening, which makes the performance of density fitting less competitive for large molecular systems, when compared to the direct method where ERIs are computed directly and can be effectively screened.

To compute B_{pq}^Q in (1), $(pq|P)$ is first computed and stored in dense matrix format where rows correspond to pq that have survived screening and columns correspond to P. Then B_{pq}^Q can be formed by dense matrix multiplication of $(pq|P)$ by $[J^{-1/2}]_{PQ}$ with optimized library functions. The resulting B_{pq}^Q is in the same dense matrix format as $(pq|P)$.

To compute V^Q and J_{pq}, we note that the expressions (2) and (3) correspond to dense matrix-vector multiplication, for which optimized library functions are also available. In (2), only the portion of D corresponding to the rs that have survived screening is needed. In (3), only the elements of J corresponding to pq that have survived screening are computed.

To compute W_{pi}^Q in (4), note that in terms of matrices, W_{pi}^Q is formed by $W(i,Q) = \sum_r C(r,i)^T B(r,Q)$ for every p. Due to shell pair screening, for a given p, only nonzero rows of $B(r,Q)$ are stored (as a dense matrix). In order to use optimized matrix multiplication library functions to compute $W(i,Q)$, for each p, we need a dense matrix consisting of the the nonzero rows of $C(r,i)$. Thus, we form these auxiliary dense matrices and use them in the optimized matrix multiplication functions.

To perform the above density fitting calculations on a distributed memory computer, GTFock partitions B_{pq}^Q, V^Q, and W_{pi}^Q along the Q dimension and distributes the calculations to each node. Partitioning along other any other dimension would require multiple large-volume communication steps. Partitioning along the Q dimension only requires reducing the partial sums of the Coulomb and exchange matrices on each node to obtain the final results.

The GTFock distributed density fitting implementation uses only uses MPI for communication and does not need to use Global Arrays or GTMatrix.

VI. TEST CALCULATIONS

Test calculations were performed using the Intel Xeon Skylake nodes on the Stampede2 supercomputer at Texas Advanced Computing Center. Each of these nodes has two sockets and 192 GB DDR4 memory, and each socket has an Intel Xeon Platinum 8160 processor with 24 cores and 2 hyperthreads per core. The interconnect system on Stampede2 is a 100 Gbps Intel Omni-Path network with a fat tree topology employing six core switches. Codes were compiled with Intel C/C++ compiler and Intel MPI version 17.0.3 with optimization flags "-xHost -O3". Intel MKL version 17.0.3 was used to perform batched dense matrix-matrix multiplication in density fitting routines.

The tolerance for shell quartet screening in GTFock was chosen to be 10^{-11}. The tolerance for primitive integral screening used inside Simint was chosen to be 10^{-14}. All reported timings for Fock matrix construction were averaged over the self-consistent field iterations needed for a Hartree–Fock calculation.

A. Direct Approach Performance

In previous work[14], we found that using Simint, ERI batching, and block buffer accumulation (Algorithm 4) for Fock matrix construction gave a 2–3 times speedup compared to not using these features. In this section, these features form the baseline for our performance comparisons. The baseline tests used Global Arrays v5.3.

The improved version of GTFock benchmarked here uses strip buffer accumulation (Algorithm 5), optimizations described in Sections III B and III C, and GTMatrix (Global Arrays is replaced compared to the baseline).

The test calculations used molecular systems derived from a protein-ligand system, 1hsg from the protein data bank. These systems consist of the ligand and a portion of its protein environment. For the three test systems called 1hsg-60 (713 atoms), 1hsg-70 (791 atoms), and 1hsg-90 (1208 atoms), all protein residues within 6, 7, and 9 Å from the ligand are included, respectively. Bonds cut by the truncation are capped appropriately[26].

Tests were performed using a moderately-contracted basis set, cc-pVDZ. In this basis set, 1hsg-60, 1hsg-70, and 1hsg-90 have 6895, 7645, and 11758 basis functions, respectively. The tests were performed using 64 nodes of the Stampede2 system described above.

Table II shows the timings for calculating the Fock matrix, comparing the baseline and improved versions. The timings are separated into its components. We observe a reduction of approximately 36% in the timings for Fock matrix accumulation and a reduction of approximately 35% in the timings for GTMatrix communication when comparing the baseline and improved versions.

Table III focuses on the communication portion of the above timings. The table shows timings for the communication procedures discussed in Section IV B and for each communication procedure we compare GTFock using Global Arrays and GTFock using the three different access modes in GTMatrix, including the new batched access mode.

TABLE II. Timings (in seconds) for GTFock baseline and improved Fock matrix construction (direct approach), separated into batched ERI calculation ("ERI"), Fock matrix accumulation ("Accum"), task scheduling ("Task"), and GTMatrix communication ("Comm").

	Procedure	GTFock Baseline	GTFock Improved
1hsg-60	ERI	5.48	5.48
	Accum	5.90	3.78
	Task	0.55	0.57
	Comm	1.15	0.76
	Total	13.08	10.59
1hsg-70	ERI	7.46	7.46
	Accum	7.60	4.82
	Task	1.12	1.05
	Comm	1.45	0.93
	Total	17.63	14.26
1hsg-90	ERI	16.48	16.48
	Accum	19.85	12.27
	Task	3.21	3.51
	Comm	3.76	1.80
	Total	42.32	34.06

The main observation is that the new batched access mode appears best for the GatherD and ScatterF procedures. The improvement for GatherD is due to the GTMatrix ring algorithm. The improvement for ScatterF is due to both the ring algorithm and the reduction of synchronization cost since each MPI process needs to post many accesses to other processes in ScatterF. As mentioned, these tests used 64 compute nodes. More benefit is expected for more nodes.

The AccFBuf procedure only involves a single target node (the node from which work was stolen or the process's own node if no work was stolen) and at most three accumulation operations, so does not benefit from batched access mode.

GTMatrix nonblocking mode has similar and sometimes better performance than Global Arrays nonblocking mode. The better performance might be explained by the extra cost of creating and maintaining handles in Global Arrays nonblocking mode, as otherwise performance mostly depends on the underlying network capabilities in this mode.

B. Density Fitting Performance

For density fitting, tests were performed using a moderately-contracted basis set, cc-pVDZ, with the auxiliary basis set cc-pVDZ-RI[52]. Tests were conducted on a set of alkane, graphene, and small protein-ligand systems shown in Table IV. We used a screening thresh-

TABLE III. Timings (in seconds) of communication procedures in Fock matrix construction using nonblocking Global Arrays operations and different access modes in GTMatrix. See Section IV B for an explanation of the communication procedures.

	Comm. Procedure	Global Arrays Nonblocking	Blocking	GTMatrix Nonblocking	Batched
1hsg-60	GatherD	0.473	0.482	0.357	0.329
	AccFBuf	0.183	0.110	0.110	0.110
	ScatterF	0.496	0.536	0.511	0.318
	Total	1.152	1.138	0.978	0.757
1hsg-70	GatherD	0.613	0.602	0.432	0.404
	AccFBuf	0.237	0.132	0.132	0.132
	ScatterF	0.603	0.664	0.620	0.391
	Total	1.453	1.398	1.184	0.927
1hsg-90	GatherD	1.277	1.334	0.864	0.861
	AccFBuf	0.256	0.248	0.248	0.248
	ScatterF	1.242	1.015	0.984	0.691
	Total	2.775	2.597	2.096	1.800

TABLE IV. Test molecular systems for density fitting.

	Atoms	Occupied Orbitals	Basis Functions	Aux. Basis Functions
Alkane-62 ($C_{20}H_{42}$)	62	81	510	1950
Alkane-122 ($C_{40}H_{82}$)	122	161	1010	3870
Alkane-182 ($C_{60}H_{122}$)	182	241	1510	5790
Alkane-242 ($C_{80}H_{162}$)	242	321	2010	7710
Alkane-302 ($C_{100}H_{202}$)	302	401	2510	9630
Alkane-362 ($C_{120}H_{242}$)	362	481	3010	11550
Graphene-72 ($C_{54}H_{18}$)	72	171	900	3834
Graphene-120 ($C_{96}H_{24}$)	120	300	1464	6696
Graphene-180 ($C_{150}H_{30}$)	180	465	2400	10350
Graphene-252 ($C_{216}H_{36}$)	252	666	3420	14796
Graphene-336 ($C_{294}H_{42}$)	336	903	4620	20034
1hsg-30 ($C_{44}H_{62}N_7O_{11}$)	124	232	1240	5022
1hsg-32 ($C_{52}H_{78}N_{10}O_{16}$)	156	295	1560	6318
1hsg-35 ($C_{69}H_{113}N_{17}O_{22}$)	221	412	2185	8823
1hsg-38 ($C_{117}H_{208}N_{31}O_{33}$)	389	696	3755	15006

old of $\sigma = 10^{-12}$ in (6).

Table V shows timings for Fock matrix construction, with and without exploiting symmetry in J and K. Timings for the direct approach are also shown, for comparison. The table includes the one-time density fitting costs: (a) computing three-index integrals ("$(pq|P)$ Build"), (b) evaluating and computing the inverse square root of the Coulomb fitting metric ("$[J^{-1/2}]$ Build"), and (c) forming B_{pq}^{Q} using Equation (1) ("B_{pq}^{Q} Build"). We note that evaluating $[J^{-1/2}]$ is not fully parallelized for running on multiple nodes, so its performance could be further optimized. The table also shows the memory usage of GTFock density fitting, which includes storing shell pair data used in Simint, storing B_{pq}^{Q}, and temporary buffers needed for holding V^{Q} and W_{pi}^{Q} when building the Fock matrix. Density fitting consumes much more memory compared to the direct approach.

The timing results show the significant advantages of the density fitting approach for small chemical systems, compared to the direct approach. As system size increases, the direct approach eventually becomes faster due to increased numerical sparsity in the ERI tensor. The results also show that the one-time costs of density fitting are significant compared to the cost of forming the Fock matrix. We note that Simint is particularly efficient in calculating the three-index integrals $(pq|P)$ since more of the calculation is efficiently vectorized than calculating the four-index ERIs[13].

We also collected timings for density fitting for these systems using the Psi4 package, which also exploits sparsity in the 3-index tensors and can be configured to use Simint. For the case of a single compute node (Psi4 does not have distributed memory density fitting) the timings are very similar to the GTFock density fitting timings when symmetry is not exploited (results not shown). This is expected as both codes are performing the same calculations in very similar ways. GTFock is faster when symmetry in J and particularly K are exploited.

Table VI, for density fitting, shows the cost of Schwarz screening overhead (computing the Schwarz bounds for (6)) and the cost of Fock matrix construction with and without screening. The results show that Schwarz screening has very small overhead compared to Fock matrix construction cost. Meanwhile, applying Schwarz screening greatly reduces the cost of evaluating (2), (3), and (4) in Fock matrix construction. The speedup due to screening is larger for larger molecules.

Figure 2 plots the speedup of Fock matrix construction using density fitting (utilizing the symmetry of J and K) compared to direct calculation. Density fitting is faster than direct calculation for small systems as opposed to large systems, and it is interesting to know where the cross-over occurs. The improved speed of the direct approach using the techniques demonstrated in this paper can make the direct approach competitive with density fitting even for systems as small as 300-400 atoms.

TABLE V. Comparison between GTFock's direct approach and density fitting approach (with and without utilizing the symmetry of J and K) for Fock matrix construction. Some tests were performed on multiple compute nodes due to memory requirements.

	Nodes	GTFock Direct		GTFock Density Fitting					
		Memory (MB)	Fock Build (s)	Memory (MB)	$(pq\|P)$ Build (s)	$[J^{-1/2}]$ Build (s)	B_{pq}^{Q} Build (s)	Fock Build (s) w/ sym.	Fock Build (s) w/o sym.
Alkane-62	1	319	0.87	2166	0.25	0.06	0.56	0.076	0.101
Alkane-122	1	741	3.43	11170	0.99	0.22	3.81	0.67	0.90
Alkane-182	1	1438	8.01	30526	1.96	0.56	11.63	2.76	4.64
Alkane-242	2	2409	6.93	68048	3.29	1.30	14.01	4.54	5.42
Alkane-302	8	3643	3.11	161152	5.25	1.65	7.08	2.89	4.34
Alkane-362	8	5161	4.28	253168	7.61	2.63	11.87	5.33	7.20
Graphene-72	1	612	7.20	13582	1.18	0.21	5.22	0.75	0.89
Graphene-120	1	1483	25.75	54070	3.41	0.77	26.37	4.79	6.86
Graphene-180	2	3262	32.05	170560	8.12	2.91	48.54	13.05	18.05
Graphene-252	8	6417	17.91	512464	17.35	5.20	39.45	13.08	17.81
Graphene-336	16	11560	17.28	1312640	32.55	18.49	51.64	20.56	28.80
1hsg-30	1	1012	9.72	24928	1.80	0.41	9.75	1.85	2.37
1hsg-32	1	1522	16.77	45732	2.94	0.69	20.53	4.32	6.13
1hsg-35	2	2802	15.54	110852	5.47	1.58	27.59	7.83	13.54
1hsg-38	8	7816	18.31	562176	19.58	5.50	38.94	16.59	20.28

TABLE VI. Timings (in seconds) of Schwarz screening overhead and density fitting Fock matrix construction with and without Schwarz screening. The symmetry of J and K are utilized. Some tests were performed on multiple compute nodes due to memory requirements.

	Nodes	Screening Overhead	Fock Build w/ screening	Fock Build w/o screeening
Alkane-62	1	0.009	0.076	0.162
Alkane-122	1	0.030	0.67	1.71
Alkane-182	4	0.078	0.76	2.35
Graphene-72	1	0.023	0.75	1.41
Graphene-120	4	0.050	1.42	3.53
1hsg-30	2	0.036	1.85	2.28
1hsg-32	4	0.060	1.26	3.56

ACKNOWLEDGMENTS

The authors would like to thank Benjamin Pritchard, Xing Liu, Jeff R. Hammond, and Min Si for assistance and helpful discussions. The authors gratefully acknowledge funding from the National Science Foundation grant ACI-1147843 and an Intel Parallel Computing Center grant. The Texas Advanced Computing Center (TACC) at the University of Texas at Austin provided HPC resources that have contributed to the research results reported within this paper.

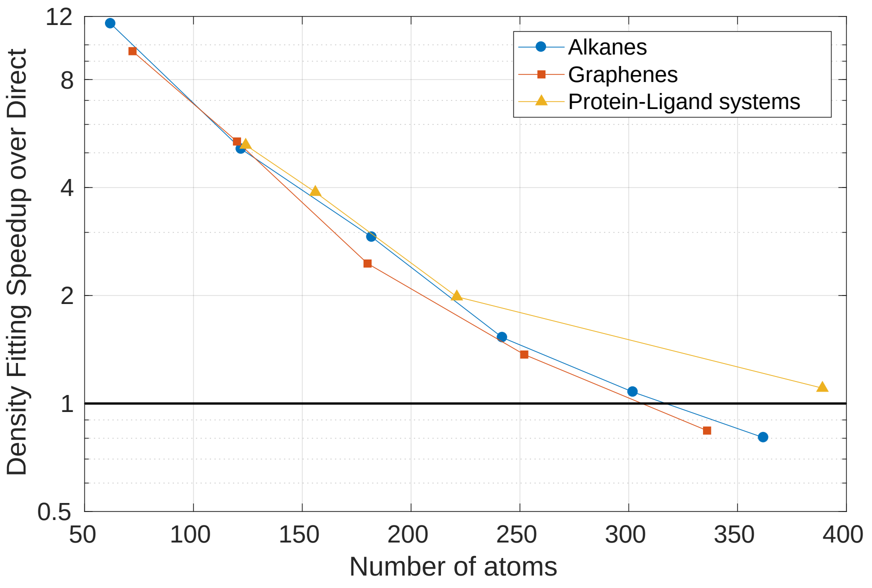

FIG. 2. Fock matrix build speedup of density fitting (utilizing the symmetry of J and K) over the direct approach in GTFock (cc-pVDZ/cc-pVDZ-RI basis sets).

REFERENCES

[1]V. R. Saunders and M. F. Guest, "Applications of the CRAY-1 for quantum chemistry calculations," Computer Physics Communications **26**, 389 – 395 (1982).

[2]P. M. W. Gill, M. Head-Gordon, and J. A. Pople, "Efficient computation of two-electron-repulsion integrals and their nth-order derivatives using contracted Gaussian basis sets," The Journal of Physical Chemistry **94**, 5564–5572 (1990).

[3]K. Wolinski, R. Haacke, J. F. Hinton, and P. Pulay, "Methods for parallel computation of SCF NMR chemical shifts by GIAO method: Efficient integral calculation, multi-Fock algorithm, and pseudodiagonalization," Journal of Computational Chemistry **18**, 816–825 (1997).

[4]K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," Journal of Computational Chemistry **29**, 334–342 (2008).

[5]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 **4**, 222–231 (2008).

[6]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 **6**, 696–704 (2010).

[7]N. Luehr, I. S. Ufimtsev, and T. J. Martinez, “Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs),” Journal of Chemical Theory and Computation **7**, 949–954 (2011).

[8]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 **32**, 2313–2318 (2011).

[9]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 **9**, 965–976 (2013).

[10]E. F. Valeev, “A library for the evaluation of molecular integrals of many-body operators over Gaussian functions,” http://libint.valeyev.net/ (2014).

[11]Q. Sun, “Libcint: An efficient general integral library for Gaussian basis functions,” Journal of Computational Chemistry **36**, 1664–1671 (2015).

[12]J. Zhang, “libreta: Computerized optimization and code synthesis for electron repulsion integral evaluation,” Journal of Chemical Theory and Computation **14**, 572–587 (2018).

[13]B. P. Pritchard and E. Chow, “Horizontal vectorization of electron repulsion integrals,” Journal of Computational Chemistry **37**, 2537–2546 (2016).

[14]H. Huang and E. Chow, “Accelerating quantum chemistry with vectorized and batched integrals,” in *SC 18': Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas, 2018).

[15]V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon, “An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor,” in *SC 17': Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado, 2017).

[16]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 **17**, 109–123 (1996).

[17]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. L. Fann, J. Nieplocha, G. S. Thomas, D. Elwood, J. L. Tilson, R. L. Shepard, A. F. Wagner, I. T. Foster, E. Lusk, and R. Stevens, “Toward high-performance computational chemistry: II. A scalable self-consistent field program,”

Journal of Computational Chemistry **17**, 124–132 (1996).

[18]T. R. Furlani, J. Kong, and P. M. W. Gill, "Parallelization of SCF calculations within Q-Chem," Computer Physics Communications **128**, 170–177 (2000).

[19]Y. Alexeev, R. A. Kendall, and M. S. Gordon, "The distributed data SCF," Computer Physics Communications **143**, 69–82 (2002).

[20]J. Baker, K. Wolinski, M. Malagoli, D. Kinghorn, P. Wolinski, G. Magyarfalvi, S. Saebo, T. Janowski, and P. Pulay, "Quantum chemistry in parallel with PQS," Journal of Computational Chemistry **30**, 317–335 (2009).

[21]K. Ishimura, K. Kuramoto, Y. Ikuta, and S.-a. Hyodo, "MPI/OpenMP hybrid parallel algorithm for Hartree–Fock calculations," Journal of Chemical Theory and Computation **6**, 1075–1080 (2010).

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

[23]H. Umeda, Y. Inadomi, T. Watanabe, T. Yagi, T. Ishimoto, T. Ikegami, H. Tadano, T. Sakurai, and U. Nagashima, "Parallel Fock matrix construction with distributed shared memory model for the fmo-mo method," Journal of Computational Chemistry **31**, 2381–2388 (2010).

[24]Y. Alexeev, A. Mahajan, S. Leyffer, G. Fletcher, and D. G. Fedorov, "Heuristic static load-balancing algorithm applied to the fragment molecular orbital method," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012) pp. 1–13.

[25]X. Liu, A. Patel, and E. Chow, "A new scalable parallel algorithm for Fock matrix construction," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (2014) pp. 902–914.

[26]E. Chow, X. Liu, S. Misra, M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X.-K. Liao, and P. Dubey, "Scaling up Hartree-Fock calculations on Tianhe-2," The International Journal of High Performance Computing Applications **30**, 85–102 (2015).

[27]T. Nakajima, M. Katouda, M. Kamiya, and Y. Nakatsuka, "NTChem: A high-performance software package for quantum molecular simulation," International Journal of Quantum Chemistry **115**, 349–359 (2015).

[28]E. Chow, X. Liu, M. Smelyanskiy, and J. R. Hammond, "Parallel scalability of Hartree-Fock calculations," The Journal of Chemical Physics **142**, 104103 (2015).

[29]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," The International Journal of High Performance Computing Applications **20**, 203–231 (2006).

[30]H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, and M. Schütz, "Molpro: a general-purpose quantum chemistry program package," Wiley Interdisciplinary Reviews: Computational Molecular Science **2**, 242–253 (2012).

[31]M. F. Guest, I. J. Bush, H. J. J. Van Dam, P. Sherwood, J. M. H. Thomas, J. H. Van Lenthe, R. W. A. Havenith, and J. Kendrick, "The GAMESS-UK electronic structure package: algorithms, developments and applications," Molecular Physics **103**, 719–747 (2005).

[32]M. Wang, A. J. May, and P. J. Knowles, "Improved version of parallel programming interface for distributed data with multiple helper servers," Computer Physics Communications **182**, 1502–1506 (2011).

[33]D. G. Fedorov, R. M. Olson, K. Kitaura, M. S. Gordon, and S. Koseki, "A new hierarchical parallelization scheme: Generalized Distributed Data Interface (GDDI), and an application to the fragment molecular orbital method (FMO)," Journal of Computational Chemistry **25**, 872–880 (2004).

[34]D. Ozog, A. Kamil, Y. Zheng, P. Hargrove, J. R. Hammond, A. Malony, W. de Jong, and K. Yelick, "A Hartree–Fock application using UPC++ and the new DArray library," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016) pp. 453–462.

[35]Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (2014) pp. 1105–1114.

[36]J. L. Whitten, "Coulombic potential energy integrals and approximations," The Journal of Chemical Physics **58**, 4496–4501 (1973).

[37]E. Baerends, D. Ellis, and P. Ros, "Self-consistent molecular Hartree–Fock–Slater calculations I. The computational procedure," Chemical Physics **2**, 41–51 (1973).

[38]B. I. Dunlap, J. W. D. Connolly, and J. R. Sabin, "On first-row diatomic molecules and

local density models," The Journal of Chemical Physics **71**, 4993 (1979).

[39]O. Vahtras, J. Almlöf, and M. Feyereisen, "Integral approximations for LCAO-SCF calculations," Chemical Physics Letters **213**, 514–518 (1993).

[40]C. D. Sherrill, "Frontiers in electronic structure theory," The Journal of Chemical Physics **132**, 110902 (2010).

[41]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, "PSI4: an open-source *ab initio* electronic structure program," Wiley Interdisciplinary Reviews: Computational Molecular Science **2**, 556–565 (2012).

[42]D. E. Bernholdt and R. J. Harrison, "Large-scale correlated electronic structure calculations: the RI-MP2 method on parallel computers," Chemical Physics Letters **250**, 477–484 (1996).

[43]H. A. Früchtl, R. A. Kendall, R. J. Harrison, and K. G. Dyall, "An implementation of RI-SCF on parallel computers," International Journal of Quantum Chemistry **64**, 63–69 (1997).

[44]L. Maschio, "Local MP2 with density fitting for periodic systems: A parallel implementation," Journal of Chemical Theory and Computation **7**, 2818–2830 (2011).

[45]T. Shiozaki, "BAGEL: Brilliantly advanced general electronic-structure library," Wiley Interdisciplinary Reviews: Computational Molecular Science **8**, e1331 (2018).

[46]S. Obara and A. Saika, "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions," The Journal of Chemical Physics **84**, 3963–3974 (1986).

[47]S. Obara and A. Saika, "General recurrence formulas for molecular integrals over Cartesian Gaussian functions," The Journal of Chemical Physics **89**, 1540–1559 (1988).

[48]H. Shan, S. Williams, W. de Jong, and L. Oliker, "Thread-level parallelization and optimization of NWChem for the Intel MIC architecture," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15 (ACM, New York, NY, USA, 2015) pp. 58–67.

[49]Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, version 3.1," Tech. Rep. (University of Tennessee, 2015).

[50]J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon,

"The design and performance of batched BLAS on modern high-performance computing systems," Procedia Computer Science **108**, 495–504 (2017).

[51]H.-J. Werner, F. R. Manby, and P. J. Knowles, "Fast linear scaling second-order Moller-Plesset perturbation theory (MP2) using local and density fitting approximations," The Journal of Chemical Physics **118**, 8149–8160 (2003).

[52]F. Weigend, A. Köhn, and C. Hättig, "Efficient use of the correlation consistent basis sets in resolution of the identity MP2 calculations," The Journal of Chemical Physics **116**, 3175–3183 (2002).