A New Scalable Parallel Algorithm for Fock Matrix Construction

Xing Liu Aftab Patel Edmond Chow
School of Computational Science and Engineering
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, 30332, USA
xing.liu@gatech.edu, aypatel@gatech.edu, echow@cc.gatech.edu

Abstract—**Hartree-Fock (HF) or self-consistent field (SCF) calculations are widely used in quantum chemistry, and are the starting point for accurate electronic correlation methods. Existing algorithms and software, however, may fail to scale for large numbers of cores of a distributed machine, particularly in the simulation of moderately-sized molecules. In existing codes, HF calculations are divided into tasks. Fine-grained tasks are better for load balance, but coarse-grained tasks require less communication. In this paper, we present a new parallelization of HF calculations that addresses this trade-off: we use fine-grained tasks to balance the computation among large numbers of cores, but we also use a scheme to assign tasks to processes to reduce communication. We specifically focus on the distributed construction of the Fock matrix arising in the HF algorithm, and describe the data access patterns in detail. For our test molecules, our implementation shows better scalability than NWChem for constructing the Fock matrix.**

I. Introduction

The Hartree-Fock (HF) algorithm, also known as the self-consistent field (SCF) algorithm, is widely used in quantum chemistry [1], and is the starting point for accurate electronic correlation methods including coupled cluster theory and many-body perturbation theory [2]. Our long-term goal is to develop HF methods suitable for a very large number of cores. In this paper, we consider the problem of the parallel distributed construction of the Fock matrix in the HF algorithm. Fock matrices also arise in Density Functional Theory (DFT) [3], another highly accurate and widely used method for electronic structure computations.

Fock matrix construction presents two main difficulties for parallelization. The first is load imbalance arising from the irregularity of the independent tasks available in the computation. The irregularity is due to the structure of molecules and a "screening" procedure used to reduce computational cost by neglecting the computation of insignificant quantities. The second difficultly is the potentially high communication cost associated with the data access pattern of Fock matrix construction. Recent proposals to use GPUs and SIMD instructions for the most expensive computations in Fock matrix construction [4], [5], [6], [7], [8], [9], [10], [11] will further make communication efficiency an important consideration in the construction of Fock matrices.

Load imbalance in HF was addressed in previous work [12], [13] by partitioning the computation into a number of independent tasks and using a centralized dynamic scheduler for scheduling them on the available processes. Communication costs are reduced by defining tasks that increase data reuse, which reduces communication volume, and consequently communication cost. This approach suffers from a few problems. First, because scheduling is completely dynamic, the mapping of tasks to processes is not known in advance, and data reuse suffers. Second, the centralized dynamic scheduler itself could become a bottleneck when scaling up to a large system [14]. Third, coarse tasks which reduce communication volume may lead to poor load balance when the ratio of tasks to processes is close to one.

We present a new scalable distributed parallel algorithm for Fock matrix construction that has significantly lower parallel overhead than other commonly used algorithms. This is achieved through the introduction of a new initial static partitioning scheme that both has reasonable load balance and increases data reuse. The scheme also enables the use of a distributed work-stealing type, dynamic scheduling algorithm for further tackling load balancing problems in a scalable manner.

There has been much recent work in parallelizing higher order approximation quantum chemistry methods, and these algorithms have been demonstrated to scale to very large numbers of cores [15], [16]. Also many tools have been developed for solving the problems inherent in the development of these types of algorithms, such as the Tensor Contraction Engine [17] and Cyclops [15]. On the other hand, the most well-known parallel algorithms for Fock matrix construction and HF are from more than a decade ago [12], [13]. Given the rapid increase in parallelism in modern supercomputers, the time has come to revisit these algorithms.

There exist many computational chemistry packages that implement parallel Fock matrix construction and HF, including NWChem [18], GAMESS [19], ACESIII [20] and MPQC [21]. We compare our algorithm to the one implemented in the NWChem, which has perhaps the most well-demonstrated scalability to large numbers of cores. Our algorithm was found to scale better, and have parallel overhead an order of magnitude less than NWChem's algorithm on test problems that were chosen to accentuate the scalability issues inherent in Fock matrix construction.

II. Background

A. Hartree-Fock Method

The Hartree-Fock method [1] approximately solves the Electronic Schrödinger equation for a molecule with a given number of electrons,

$$\hat{H}_{elec}\Psi_{elec} = E\Psi_{elec}.$$

In this expression Ψ_{elec} is a called a *wavefunction*, E is the energy of the wavefunction, and $\hat{H}_{elec}$ is a given differential operator known as the electronic Hamiltonian. The quantum mechanical information about the molecule is contained in its wavefunction, which can be used to predict its chemical and physical properties.

Ψ_{elec} can be expressed in terms of a set of functions $\psi_i(\mathbf{r})$ on $\mathbb{R}^3$, called molecular orbitals [1]. A molecule is said to have *closed shells* if the electrons in it are all paired, i.e. the number of electrons in the molecule, n, is even. The number of *occupied* molecular orbitals associated with a closed shell molecule is then $n/2$. We only consider the HF method for closed shell molecules in this paper.

The Hartree Fock method is a variational method that computes a finite basis approximation to the molecular orbitals of a given molecule. This is an approximation of the form

$$\psi_i(\mathbf{r}) = \sum_{j=1}^{n_f} c_{ij}\phi_j(\mathbf{r}), \tag{1}$$

where the functions $\phi_j(\mathbf{r})$ are functions on $\mathbb{R}^3$, called *basis functions*, and are chosen to model the physical properties of the molecule. Commonly, the set of basis functions chosen for computing the molecular orbitals of a particular molecule is divided into groups called *shells*. Each shell of basis functions shares the same quantum angular momentum and the same *center*. Typically, the centers of shells correspond to the atomic coordinates of the atoms of the molecule. A basis set can also be divided into sets of *atoms*, which are sets of shells that have the same center. We note here that the terms basis function, shell, and atom are also commonly used to refer to the indices used to represent them in a computer program. We adopt this convention throughout this paper. In all cases, what we are referring to should be clear from context.

B. Hartree-Fock Algorithm

The Hartree-Fock algorithm is an iterative method for computing the coefficients c_{ij} in equation (1). It takes, as its input, the coordinates, in $\mathbb{R}^3$, of the atoms of the molecule, the atomic numbers of the atoms, a basis set type, and an initial guess. As its output, it produces three matrices, a *Fock matrix F*, a *density matrix D*, and a coefficient matrix C. The coefficients c_{ij} in equation (1) are the elements of C. The basis set type is nothing but a rule that, along with the other inputs, forms a complete specification of the basis functions ϕ_j in equation (1).

Each iteration k of the HF algorithm has two main computational steps, the formation of the kth approximation to the Fock matrix F, using the $(k-1)$th density matrix D, and the diagonalization of this matrix to obtain the kth approximation to D. The iteration is usually stopped when the magnitude of the difference in values between the current density matrix and the previous density matrix has fallen below a certain, pre-chosen, convergence threshold. This is the origin of the term "self-consistent field."

Algorithm 1: HF Algorithm

```
1  Guess $D$
2  Compute $H^{core}$
3  Diagonalize $S = UsU^T$
4  Form $X = Us^{1/2}$
5  repeat
6      Construct $F$
7      Form $F' = X^T F X$
8      Diagonalize $F' = C'\varepsilon C'^T$
9      $C = XC'$
10     Form $D = 2C_{occ}C_{occ}^T$
11 until converged
```

The HF algorithm is presented in Algorithm 1. Here, C_{occ} is the portion of the C matrix corresponding to the smallest $n/2$ eigenvalues of F', where n is the number of electrons of the molecule. The matrix S is called the overlap matrix, X is a basis transformation matrix computed using S, and H_{core} is a matrix called the *core Hamiltonian*. The matrices S,X, and H_{core} do not change from one iteration to the next and are usually precomputed and stored.

C. Fock Matrix Construction

We focus on the formation of F in line 6 of Algorithm 1, and the data access pattern required. This step involves a four dimensional array of numbers, each of whose entries is denoted by $(ij|kl)$ where the indices i,j,k,l run from 0 to $n_f - 1$, where n_f is the number of basis functions used. Each $(ij|kl)$ is a six dimensional integral, called an electron repulsion integral (ERI), given by

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

where the x_1 and x_2 are coordinates in $\mathbb{R}^3$, the ϕ's are basis functions, $r_{12} = \|x_1 - x_2\|$, and the integral is over all of $\mathbb{R}^3 \times \mathbb{R}^3$. In terms of these quantities and the density matrix D, the expression for an element of the Fock matrix is

$$F_{ij} = H_{ij}^{core} + \sum_{kl} D_{kl}\left(2(kl|ij) - (ik|jl)\right). \tag{3}$$

The ERIs possess permutational symmetries, given by

$$(ij|kl) = (ji|kl) = (ij|lk) = (ji|lk) = (kl|ij). \tag{4}$$

Hence, the number of unique ERIs is $\approx n_f^4/8$. It is prohibitively expensive to precompute and store the ERIs in memory for all but the smallest of molecules; they must be recomputed each time a Fock matrix is constructed, which is once per iteration of the HF algorithm. ERI computation is expensive, making Fock matrix construction a significant portion of the runtime of the HF algorithm.

An important optimization in the computation of ERIs is to reuse intermediate quantities shared by basis functions in a shell. Thus, almost all algorithms for ERI computation compute ERIs in batches called *shell quartets*, defined as

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

These batches are 4-dimensional arrays of different sizes and shapes. The fact that these irregular batches are the minimal units of work is the main reason for the great complexity of efficient parallel HF codes.

Referring to equation (3), we introduce the *Coulomb* and *exchange* matrices, J and K,

$$J_{ij} = \sum_{kl} D_{kl}(kl|ij),$$

$$K_{ij} = \sum_{kl} D_{kl}(ki|lj).$$

In terms of these, equation (3) becomes

$$F = H^{core} + 2J - K.$$

We defined J and K to emphasize the fact that they require *different* parts of the array of ERIs. Using Matlab-like indexing notation, J_{ij} requires $(ij|:,:)$, and K_{ij} requires $(i,:|j,:)$. Thus, if we want to exploit the symmetry of the ERIs and only compute unique ERIs, we need to phrase the construction of F in terms of the computation of the integrals $(ij|kl)$ rather than each F_{ij}.

In practice, as each shell quartet of integrals is computed, the corresponding blocks of F are updated. These are in different locations for the contributions arising through J, and K, resulting in a highly irregular pattern of accesses to D and F. For a shell quartet, $(MN|PQ)$, the blocks of F that need to be updated are F_{MN} and F_{PQ} for J, and F_{MP}, F_{NQ}, F_{MQ} and F_{NP} for K. Further, these updates require blocks D_{PQ}, D_{NQ}, D_{MN}, D_{MP}, D_{NP}, and D_{MQ} of D. Thus, for each shell quartet of integrals computed, six shell blocks of D need to be read and six shell blocks of F are updated. Note here that we use shell indices to denote the blocks of F and D corresponding to the basis function indices of the basis functions in the shells. This is common practice since indexing is arbitrary, and basis functions are usually indexed such that all the basis functions associated with a given shell are consecutively numbered. Thus, F_{MN} is nothing but the block,

$$F_{MN} = \{F_{ij} \text{ s.t. } i \in \text{shell } M, j \in \text{shell } N\}.$$

D. Screening

It turns out that many of the $n_f^4/8$ ERIs are zero or negligibly small. This is a consequence of the fact that the integral in equation (2) is small if the centers of the pair of the ϕ to the left or right of r_{12} are centered at points in space that are physically far from each other. More exactly, we have the relation [22]

$$(ij|kl) \leq \sqrt{(ij|ij)(kl|kl)}.$$

Thus, if we have determined a drop tolerance τ for $(ij|kl)$ that yields the required accuracy of the computed F and D from the HF algorithm, we can neglect the computation of the integrals $(ij|kl)$ for which

$$\sqrt{(ij|ij)(kl|kl)} < \tau. \tag{5}$$

The use of relation (5) to drop integrals is a procedure called *Cauchy-Schwarz screening*, and it can be shown that the number of integrals remaining after applying it is significantly less than $n_f^4/8$, especially for large molecules [22]. Thus, for computational efficiency, it is essential to utilize screening.

As mentioned in Section II-C, integrals are computed in batches called shell quartets. We require a few definitions to illustrate how screening is applied to shell quartets. The *shell pair value* of a pair of shells is

$$(MN) = \max_{i,k \in M j,l \in N} (ij|kl).$$

In practice, the shell pair values are usually computed and stored. Then, we can skip computation of a shell quartet $(MN|PQ)$ if $\sqrt{(MN)(PQ)} < \tau$. There is also the associated concept of *significance*. A shell pair MN is significant if

$$(MN) \geq \tau/m^*$$

where, for a basis set $\mathcal{B}$, m^* is defined as

$$m^* = \max_{M,N \in \mathcal{B}} (MN).$$

It is also a common practice to compute integrals in larger batches called *atom quartets*. Recalling that an atom corresponds to a set of shells with the same center, an atom quartet $(I_{at}J_{at}|K_{at}L_{at})$, where $I_{at}, J_{at}, K_{at}, L_{at}$ are atoms, is defined as

$$(M_{at}N_{at}|P_{at}Q_{at}) = \{(MN|PQ) \text{ s.t. } M \in M_{at},\ N \in N_{at},\ P \in P_{at},\ Q \in Q_{at}\}.$$

Screening can also be applied to atom quartets of integrals by an obvious extension of the procedure for shell quartets. Also, the concepts of significance and pair values extend to atom pairs trivially.

E. Notes on Parallelization

In Section I we mentioned that the two main difficulties of parallel Fock matrix construction are load imbalance, due to the irregularity of the independent tasks of the computation, and the high communication cost associated with the irregular data access pattern.

The computationally expensive part of Fock matrix construction is the computation of the ERIs. These can only be computed in batches of shell quartets, which may not be of the same size for different shells. Further, for large problems, many shell quartets are dropped by screening. In addition, even different shell quartets of ERIs with the same number of elements may take different times to compute. These factors make it hard to obtain an initial balanced partitioning of the computational volume of Fock matrix construction.

Further, to tackle large problems we need to distribute the D and F matrices among the processors of a distributed system. The irregular pattern of access to this data, arising from the updates that need to be made as each shell quartet of ERIs is computed, generates communication. Thus, in order to reduce communications costs, it is desirable to have algorithms that maximize the reuse of data that is local or has already been transferred to a given process.

F. Previous Work

The NWChem computational chemistry package [12] distributes the matrices F and D in block row fashion among the available processes. The indices of the basis set are grouped by *atoms*, and if there are n_{atoms} atoms then process i owns the block rows of F and D corresponding to atom indices ranging from $(i \cdot n_{atoms}/p)$ to $((i+1) \cdot n_{atoms}/p) - 1$, where p is the total number of processes used. Note that in the above, we have assumed that p divides n_{atoms}.

Further, a task-based computational model is used in order to utilize the full permutational symmetry of the integrals. Each task is defined as the computation of 5 atom quartets of integrals, the communication of the parts of D corresponding to these blocks of integrals, and the updating of the corresponding parts of F. The choice of atom quartets as minimal computational units is made in order to increase data locality and reduce communication volume. Using Matlab-like notation, the task definition used in [12] is $(I_{at}J_{at}|K_{at}, L_{at} : L_{at}+4)$, where the $I_{at}, J_{at}, K_{at}, L_{at}$ are atom indices. This choice is a compromise between fine task granularity and low communication volume through data reuse [12], [13].

For load balancing, the tasks are dynamically scheduled on processes using a simple centralized dynamic scheduling algorithm. Processes extract tasks from a centralized task queue and execute them. Screening and symmetry consideration are incorporated into the task execution process, and only the unique significant shell quartets of integrals are computed.

This approach suffers from three problems. Firstly, the use of 5 atom quartets as a minimal unit of work does not allow for fine enough granularity when large numbers of processes are used, and as a result, load balancing suffers. It would seem at this point that a choice of a smaller unit of work could solve this problem, however the communication volume, and consequently the communication cost is also likely to increase if this is done. Secondly, the task scheduling is completely dynamic, with no guarantee of which tasks get executed on which processes, so it is not possible to prefetch all the blocks of D required by a processes in a single step, before starting integral computation. Lastly the centralized dynamic scheduler is likely to become a bottleneck in cases when large numbers of cores are used.

The precise details of the algorithm are presented in Algorithm 2.

Algorithm 2: FockBuild

```
On process $p$ do
task = GetTask()   /*Global task queue access*/
id = 0
for Unique triplets $I_{at}, J_{at}, K_{at}$ do
    if $(I_{at}J_{at})$ is significant then
        $l_{hi} = K_{at}$
        if $K_{at} = I_{at}$ then
            $l_{hi} = J_{at}$
        end
        for $l_{lo}$ from 1 to $l_{hi}$ stride 5 do
            if id = task then
                for $L_{at}$ from $l_{lo}$ to $\min(l_{lo}+4, l_{hi})$ do
                    if $(I_{at}J_{at})(K_{at}L_{at}) > \tau^2$ then
                        Fetch blocks of $D$
                        Compute $(I_{at}J_{at}|K_{at}L_{at})$
                        Update blocks of $F$
                    end
                    id = id +1
                end
                task = GetTask()
            end
        end
    end
end
```

III. New Algorithm for Parallel Fock Matrix Construction

A. Overview

Our algorithm reduces communication costs while simultaneously tackling the problem of load balance by using an initial static task partitioning scheme along with a work-stealing distributed dynamic scheduling algorithm. The reduced communication, along with the better scalability of work-stealing type scheduling algorithms [14], gives it better scalability than existing approaches.

We first specify an initial static task partitioning scheme that has reasonable load balance. The initial static partitioning allows us to know approximately on which processes tasks are likely to get executed, which in turn, allows us to perform all the communication for each process in a few steps. We also reorder shells in a basis set to increase overlap in the data that needs to be communicated by the tasks initially assigned to each process, which leads to a reduction in communication volume and hence communication cost.

B. Task Description

To describe the computation associated with tasks in our algorithm, we need the concept of what we call the *significant set* of a shell. Recalling the definition of significance from Section II-D, we define the significant set of a shell M to be the set of all the shells N such that the pair MN is significant. More formally, this is the set

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

where τ and m^* retain their definitions from Section II-D.

Also, we define the set $(M,:|N,:)$ corresponding to the shells M and N in the basis set, denoted by $\mathcal{B}$ to be,

$$(M,:|N,:) = \{(MP|NQ) \text{ s.t. } P \in \mathcal{B},\ Q \in \mathcal{B}\}.$$

Now, a task is defined as the computation of the integrals $(M,:|N,:)$, and the updating of the corresponding blocks of the F matrix using the appropriate blocks of D. One can simply see that, after screening, $(M,:|N,:)$ contains $|\Phi(M)||\Phi(N)|$ significant shell quartets. That is,

$$(M,:|N,:) = \{(MP|NQ) \text{ s.t. } P \in \Phi(M),\ Q \in \Phi(N)\}.$$

From equation (3), the parts of D that need to be read, and the parts of the Fock matrix F that need to be updated are the shell blocks $(M,\Phi(M)),(N,\Phi(N)),(\Phi(M),\Phi(N))$. It can be seen that the six blocks of F and D associated with the computation of each shell quartet in a task are all contained within these parts of F and D. Further, for this definition of a task, the maximum number of tasks available for a problem with n_{shells} shells is n_{shells}^2.

Whether or not a pair of shells M and N is significant as defined in Section II-D is related to the distance between their centers, i.e., the distance between the atomic coordinates of the atoms that they are associated with. This, in a certain sense, implies that for a molecule whose atomic coordinates are distributed more or less uniformly in a contiguous region of space, the variation in $|\Phi(M)|$, for different shells M, should not be too large. This in turn implies that, for different shell pairs MN, the variation in $|\Phi(M)||\Phi(N)|$ should not be too large either. Thus, the amount of integral computation associated with different tasks should not vary widely.

C. Initial Static Partitioning

The tasks are initially equally distributed among processes. If we have a $p_{row} \times p_{col}$ rectangular virtual process grid, for a problem with n_{shells} shells in the basis set, tasks are initially assigned to processes in blocks of size $n_{br} \times n_{bc}$, where $n_{br} = n_{shells}/p_{row}$, and $n_{bc} = n_{shells}/p_{col}$. That is, the process p_{ij} is initially assigned the block of tasks corresponding to the computation of the set of shell quartets $(i \cdot n_{br} : (i+1) \cdot n_{br} - 1,: | j \cdot n_{bc} : (j+1) \cdot n_{bc} - 1,:)$. For simplicity, we have again assumed that p_{row} and p_{col} divide n_{shells}. From the comment at the end of the previous section, we expect that the time for integral computation for each task is approximately the same.

Having this initial static partitioning, each process can now prefetch the parts of D associated with the tasks assigned to it, and store them in a local buffer D_{local}. Also, a local buffer to hold the updates to F, F_{local}, is initialized before beginning the execution of integral computation. Subsequently, as shell quartets of integrals are computed, the process uses data in D_{local} to update F_{local}.

On the surface it would seem that we do not consider the permutational symmetry of the ERIs as per equation (4). However, with our task description we can enforce computation of only the unique integrals by computing a shell quartet only if certain relationships between its indices are satisfied. Consider a subroutine *SymmetryCheck*(M,N) for integers M, N, that returns *true* if either of the conditions, "$(M > N)$ *and* $(M+N)$" or "$(M \leq N)$ *and* $(M+N)$ is odd", is satisfied, and returns *false* otherwise. Computation of only unique shell quartets can be enforced using this on pairs of indices. Now we can give a complete specification of the operations performed in a task. This is presented in Algorithm 3.

Algorithm 3: doTask $(M,:|N,:)$

```
1 for Q from 0 to $n_{shells}-1$ do
2     for P from 0 to $n_{shells}-1$ do
3         if SymmetryCheck$(M,N)$ and SymmetryCheck$(M,P)$
          and SymmetryCheck$(N,Q)$ and $(MN)(PQ) > \tau$ then
4             Compute $(MP|NQ)$
5             Update blocks of $F_{local}$, using $D_{local}$
6         end
7     end
8 end
```

Note that in the above τ is the tolerance chosen for screening. Once computation is finished, the local F buffers can then be used to update the distributed F matrix.

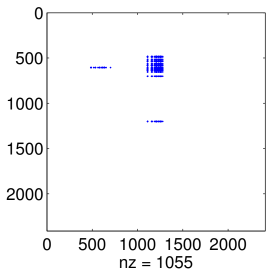
D. Shell Reordering

The parts of D that need to be read, and F that need to be updated by a task that computes the integrals $(M,:|N,:)$ are given by the index sets $(M,\Phi(M)),(N,\Phi(N)),(\Phi(M),\Phi(N))$. As explained in the previous section, these parts corresponding to a block of tasks assigned to a process are prefetched.

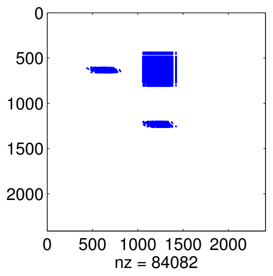
Naturally, in order to reduce latency costs associated with this prefetching, we would like these regions of F and D to be as close in shape as possible to contiguous blocks. This would happen if $\Phi(M)$ and $\Phi(N)$ are such that the difference between the maximum and minimum shell indices in these sets is small. This can be achieved if pairs of shells that are significant have indices that are close together. Since the indexing of shells is arbitrary, and a shell pair is more likely to be significant if the distance between the centers of the pair is small, we could achieve this approximately by choosing an indexing scheme that numbers shells, whose centers are in close spatial proximity, similarly.

In our algorithm we utilize an initial shell ordering that does this to a certain extent. First, we define a three dimensional cubical region that contains the atomic coordinates of the molecule under consideration. This region is then divided into small cubical cells, which are indexed using a natural ordering. Then shells are ordered with those appearing in consecutively numbered cells being numbered consecutively, with the numbering within a cell being arbitrary. The basis function numbering is chosen so that basis functions within a shell are consecutively numbered, and the basis functions in two consecutively numbered shells form a contiguous list of integers.

This reordering has another very desirable consequence. In the initial partitioning scheme that we use, each process is assigned tasks that correspond to a block of shell pairs. As a result of this, once our shell ordering has been applied there is considerable overlap in the elements of F and D that need to be communicated by the tasks assigned to a process. This is illustrated by Figure 1 of which part (a) shows the parts of the density matrix D, and the number of elements, required by $(300,:|600,:)$, and (b) shows the parts required by the task block $(300:350,:|600:650,:)$ for the molecule $C_{100}H_{202}$ which has, with the cc-pVDZ basis set, 1206 shells and 2410 basis functions. The number of tasks in the latter block is 2500, however, the number of elements of D required is only about 80 times greater than that for the former single task.



(a) $(300,:|600,:)$



(b) $(300:350,:|600:650,:)$

Figure 1: Map of elements of D required by (a) $(300,:|600,:)$ and (b) $(300:350,:|600:650,:)$.

E. Algorithm

Our algorithm assumes that both D and F are stored in distributed fashion, using a 2D blocked format. This is necessary when n_{shells} is large. The process p_{ij} in the process grid owns the shell blocks of F and D corresponding to the shell pair indices $(i\cdot n_{br}:(i+1)\cdot n_{br}-1, j\cdot n_{bc}:(j+1)\cdot n_{bc}-1)$, where the definitions of n_{br} and n_{bc} are the same as those in Section III-C.

At the beginning of execution, each process populates a local task queue with the tasks assigned to it according to the static partition described in Section III-C. It then prefetches all the blocks of D required by its tasks from the distributed D matrix and stores them in a contiguous buffer in local memory. A local buffer of appropriate size to hold updates to F for the tasks assigned to the process is also initialized. Subsequently, each task is extracted from the queue and executed. As explained in Section III-B, this involves the computation of the shell quartets assigned to it, and the updating of the corresponding shell blocks of F using blocks of D. Because data has been prefetched, updates are performed to the local contiguous buffers. Once all the tasks are complete, the local F buffers are used to update the distributed F matrix. This is presented as Algorithm 4.

Algorithm 4: FockBuild for process p_{ij}

```
1 Initialize task queue Q
2 Populate task queue with tasks
    $(i\cdot n_{br}:(i+1)\cdot n_{br}-1,:|j\cdot n_{bc}:(j+1)\cdot n_{bc}-1,:)$
3 Fetch and store required $D$ blocks in $D_{local}$
4 Initialize $F_{local}$
5 while NotEmpty(Q) do
6     $(M,:|N,:) \leftarrow$ ExtractTask(Q)
7     doTask $(M,:|N,:)$
8 end
9 Update $F$ using $F_{local}$
```

In practice, all communication operations are performed using the Global Arrays framework [23], which provides one-sided message passing operations, and is used to phrase communication in a manner similar to data access operations in shared memory.

F. Work-Stealing Scheduler

In spite of the fact that our initial partitioning scheme assigns blocks of tasks that have *similar* computational costs to process, it is not perfectly balanced. Dynamic load-balancing is required, which we achieve through the use of a simple work-stealing distributed dynamic scheduling algorithm [24]. This is implemented using Global Arrays.

When the task queue on a process becomes empty, it scans the processes in the process grid in a row-wise manner, starting from its row, until it encounters one with a non-empty task queue. Then it steals a block of tasks from this victim processor's task queue and adds it to its own queue, updating the victim's queue during this process. After this, it copies the local D buffer of the victim to its local memory and initializes a local buffer for updates to F corresponding to this, and updates these buffers during the execution of the stolen tasks. When a process steals from a new victim the current stolen F buffer is accumulated to F_{local} of the previous victim.

G. Performance Model and Analysis

In order to develop a model for the average running time of our algorithm, we have to make a few simplifying assumptions and define a few terms. The average time taken to compute an ERI is denoted by t_{int}. A square process grid is assumed with $p_{row} = p_{col} = \sqrt{p}$, p being the total number of processes used. It is also assumed that p divides n_{shells}. The average number of basis functions associated with a shell is denoted by A, and the average number of shells in the set $\Phi(M)$, for a shell M, is denoted by B. The average number of elements in $\Phi(M) \cap \Phi(M+1)$ for shell M is q. We also assume that the average number of processors from which tasks are stolen by a given process, using the algorithm described in Section III-F, is s. Lastly, the bandwidth of the communication network of the distributed system is taken to be β.

The computation cost of Fock matrix construction is the cost of computing the ERIs. The number of shell quartets in $(M,:|N,:)$ after screening is $|\Phi(M)||\Phi(N)|$, which has average value B^2. Thus, the number of shell quartets associated with the block of tasks assigned to each processor under the initial static partitioning scheme described in Section III-C is $n_{shells}^2 B^2/p$. Further, we only compute unique shell quartets of integrals. Thus, the number of shell quartets assigned to a process becomes $n_{shells}^2 B^2/8p$. Now, using A and t_{int}, the expression for average compute time is

$$T_{comp}(p) = \frac{t_{int} B^2 A^2 n_{shells}^2}{8p}. \tag{6}$$

We recall from Section III-B that the task associated with the integrals $(M,:|N,:)$ needs to communicate shell blocks of F and D with shell pair indices $(M,\Phi(M))$, $(N,\Phi(M))$ and $(\Phi(M),\Phi(N))$. Each process p_{ij} under the initial static partitioning owns the task block corresponding to the integrals $(i\cdot n_b : (i+1)\cdot n_b - 1,: | j\cdot n_b : (j+1)\cdot n_b - 1,:)$, where $n_b = n_{shells}/\sqrt{p}$. Thus, the average communication volume for a process arising from the need to communicate blocks of F and D, corresponding to the blocks $(M,\Phi(M))$ and $(N,\Phi(N))$, for shells M and N in $(i\cdot n_b : (i+1)\cdot n_b - 1)$ and $(j\cdot n_b : (j+1)\cdot n_b - 1)$, respectively, is

$$v_1(p) = 4A^2 B n_{shells}^2/p. \tag{7}$$

The communication volume associated with the blocks $(\Phi(M),\Phi(N))$ is a little more tricky to obtain since we have to take into consideration the overlap in these sets for the tasks associated with a process, as explained in Section III-D. If $\Phi(M)$ and $\Phi(M+1)$ have q elements in common, and the average value of $|\Phi(M)|$ is B, then the average number of elements in $\Phi(M+1)\cup\Phi(M) - \Phi(M)\cap\Phi(M+1)$ is $2(B-q)$. Thus, $|\Phi(M)\cup\Phi(M+1)|$ should be $q+2(B-q)$. This can be extended to $n_{shells}/\sqrt{p}$ shells, giving the expression $(q + (n_{shells}/\sqrt{p})(B-q))$ for the average number of elements in the union of the Φ sets corresponding to these shells. Thus, the average communication volume arising from these sets for a process is

$$v_2(p) = 2\left(\frac{n_{shells}}{\sqrt{p}}(B-q)+q\right)^2 A^2. \tag{8}$$

The average communication volume, including the communication for D and F buffers for steals, is

$$V(p) = (1+s)\left(v_1(p)+v_2(p)\right). \tag{9}$$

Now, an expression for the communication time is

$$T_{comm}(p) = \frac{1}{\beta}V(p). \tag{10}$$

With equations (6) and (10), we are now in a position to arrive at an expression for the *efficiency* of the new parallel algorithm. Efficiency is given by

$$E(p) = \frac{T^*}{pT(p)},$$

where T^* is the *running* time of the fastest sequential algorithm for solving the same problem as the parallel algorithm. In our case, since we utilize screening, and only compute unique ERIs, we make the assumption that $T^* = T_{comp}(1)$. Also, $pT_{comp}(p) = T_{comp}(1)$, and we assume no overlap in computation and communication, so $T(p) = T_{comp}(p) + T_{comm}(p)$, yielding

$$E(p) = \frac{1}{1+T_{comm}(p)/T_{comp}(p)}.$$

Thus efficiency depends on the ratio $T_{comm}(p)/T_{comp}(p)$, which we denote by $L(p)$. Using equations (7), (8), (9), (10) and (6) and some trivial algebraic manipulations we get

$$L(p) = \frac{16(1+s)}{t_{int}B^2\beta}\left(\left((B-q)+q\frac{\sqrt{p}}{n_{shells}}\right)^2 + 2B\right). \tag{11}$$

Expression (11) tells us several things, the first being the *isoefficiency function* of our algorithm, which is defined as the rate at which the problem size must vary in terms of the number of processes, in order to keep efficiency constant. Efficiency is constant in our case if L is constant and L is constant if $\sqrt{p}/n_{shells}$ is constant, assuming that s does not vary with p. This gives us an isoefficiency function of $n_{shells} = O(\sqrt{p})$. Hence, in order for us to have constant efficiency, the problem size, specified in terms of the number of shells, must grow at least as fast as $\sqrt{p}$.

In the above analysis we have assumed s to be a constant. This is expected to be true if both p and n_{shells} (and hence the amount of computational work) are increasing.

Equation (11) also gives us some other qualitative information. Substituting $p = n_{shells}^2$, the maximum available parallelism, we obtain

$$L(n_{shells}^2) = \frac{16(1+s)}{t_{int}\beta}\left(1+\frac{2}{B}\right). \quad (12)$$

Now, with increasing number of processes, the algorithm will only reach a point at which communication starts to dominate if this is greater than 1. This is likely to happen sooner as t_{int} goes down, with improvements in integral calculation algorithms and technology. Further, for highly heterogeneous problems with many widely varying atom types that are irregularly distributed in space, we expect that our initial task partitioning will be less balanced, implying that the number s is likely to go up, making communication dominate sooner.

The presence of the term $2/B$ tells us about the effect that the structure of the molecule has on the running time. B is the average value of $|\Phi(M)|$, which is the number of shells that have a significant interaction with M, and this number is expected to be very large for a molecule, that has atoms centered at points that are densely distributed in three dimensional space. This larger value indicates, from expression (12), that computation dominates for such a problem, as expected.

Expression (12) can be used to determine how much smaller t_{int} needs to be for there to exist a point at which communication costs start to dominate. Consider the molecule $C_{96}H_{24}$. With the cc-pVDZ basis set it was observed that, using 3888 cores on a test machine (described in Section IV), the average value for s for our Fock matrix construction algorithm was 3.8. For simplicity we assume that, B is large so that $2/B \approx 0$ for this problem. Also, the bandwidth of the interconnect of the test machine was 5 GB/s. Using $t_{int} = 4.76\mu s$ from Table V in Section IV and expression (12) we can arrive at the conclusion that integral computation has to be approximately 50 times faster for there to exist a point at which communication starts to dominate. This is supported by the results in Figure 2, which indicate that this case is still heavily dominated by computation with 3888 cores. In contrast to this, for NWChem's algorithm described in Section II-F, the parallel overhead time, of which communication cost is a component, actually becomes greater than the computation time at $p \approx 3000$ (refer to Figure 2).

In all the analysis in this section, we have made no mention of the latency costs associated with communication. We do this for simplicity. All that can be said is that the latency costs will add to the communication time, increasing $L(p)$ and reducing the critical number of processes at which communication costs surpass computation costs.

IV. Experimental Results

A. Experimental Setup

Tests were conducted on Lonestar [25] located at the Texas Advanced Computing Center. Table I shows the configuration of the machine.

Table I: Machine parameters for each node of Lonestar.

Component	Value
CPU	Intel X5680
Freq. (GHz)	3.33
Sockets/Cores/Threads	2/12/12
Cache L1/L2/L3 (KB)	64/256/12288
GFlop/s (DP)	160
Memory (GB)	24

The nodes are connected by an InfiniBand Mellanox switch with a bandwidth of 5 GB/s. The normal queue was used which allows a maximum number of 4104 cores (342 nodes) to be requested.

The implementation of our algorithm, which we refer to as GTFock, was compared to the distributed Fock matrix construction algorithm implemented in NWChem version 6.3 [13], [12]. Our implementation uses Global Arrays [23] version 5.2.2, which is the same version used by NWChem 6.3. We also use the MPICH2 version 2.1.6 implementation of MPI-2. For ERI computation, the ERD integrals package [26] is used. This is distributed as part of the ACES III computational chemistry package [20]. Intel compilers ifort version 11.1 and icc version 11.1 were used to compile both NWChem and GTFock, and both were linked against the Intel MKL version 11.1 libraries.

Our implementation uses OpenMP multithreading to parallelize the computations associated with a task, given in Algorithm 3. Consequently, we ran GTFock with one MPI process per node. NWChem does not use multithreading, and one MPI process per core was used in this case.

We used four test molecules with the Dunning cc-pVDZ basis set [27]. The test cases along with their properties are presented in Table II. The first two molecules have a 2D planar structure similar to the carbon allotrope, graphene. The latter two are linear alkanes, which have a 1D chain-like structure. A screening tolerance of $\tau = 10^{-10}$ was used for all tests, for both our implementation and for NWChem, and for a fair comparison, optimizations related to symmetries in molecular geometry were disabled in NWChem.

Table II: Test molecules.

Molecule	Atoms	Shells	Functions	Unique Shell Quartets
$C_{96}H_{24}$	120	648	1464	1.19×10^9
$C_{150}H_{30}$	180	990	2250	3.12×10^9
$C_{100}H_{202}$	302	1206	2410	1.68×10^9
$C_{144}H_{290}$	434	1734	3466	3.52×10^9

B. Performance of Fock Matrix Construction

Table III compares the running time for Fock matrix construction for NWChem and GTFock for the test cases just described. Although NWChem is faster for smaller core counts, GTFock is faster for larger core counts. Table IV

Table III: Fock matrix construction time (in seconds) for GTFock and NWChem on four test cases. Although NWChem is faster for smaller core counts, GTFock is faster for larger core counts.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	673.07	649.00	1765.92	1710.00	619.15	406.00	1290.58	845.00
108	75.17	75.00	196.95	198.00	68.32	48.00	140.14	97.00
192	42.53	42.00	111.02	115.00	38.58	27.00	78.92	57.00
768	10.78	12.00	28.03	29.00	9.72	7.40	19.91	15.00
1728	4.93	5.30	12.57	13.00	4.37	4.80	9.03	7.30
3072	2.91	4.10	7.21	8.50	2.50	5.10	5.11	5.30
3888	2.32	4.50	5.80	6.70	2.02	5.80	4.06	9.00

Table IV: Speedup in Fock matrix construction for GTFock and NWChem on four test cases, using the data in the previous table. Speedup for both GTFock and NWChem is computed using the fastest 12-core running time, which is from NWChem. GTFock has better speedup at 3888 cores.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	11.57	12.00	11.62	12.00	7.87	12.00	7.86	12.00
108	103.60	103.84	104.19	103.64	71.31	101.50	72.36	104.54
192	183.10	185.43	184.84	178.43	126.27	180.44	128.48	177.89
768	722.72	649.00	732.15	707.59	501.44	658.38	509.22	676.00
1728	1581.00	1469.43	1631.94	1578.46	1114.87	1015.00	1122.43	1389.04
3072	2678.13	1899.51	2847.23	2414.12	1949.58	955.29	1983.57	1913.21
3888	3354.01	1730.67	3540.98	3062.69	2415.47	840.00	2498.77	1126.67

Table V: Average time, t_{int}, for computing each ERI for GTFock (using the ERD library) and NWChem.

Mol.	Atoms/Shells/Funcs	t_{int} GTFock	t_{int} NWChem
$C_{24}H_{12}$	36/180/396	4.759 μs	3.842 μs
$C_{10}H_{22}$	32/126/250	3.060 μs	2.400 μs

shows the corresponding speedups and shows that GTFock has better scalability than NWChem up to 3888 cores.

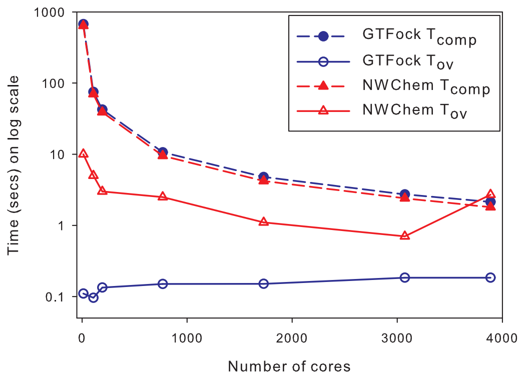
We used the NWChem running time on a single node to compute the speedup for both NWChem and GTFock, since NWChem is faster on a single node. NWChem's better single-node performance is most likely due to its better use of *primitive pre-screening* [26] to avoid computation of negligible contributions to integrals. The results in Table V compare the performance of the integral packages of both implementations on a machine with similar characteristics as one node of our test machine, for two molecules that are representative, structurally, of the molecules that we used to test Fock matrix construction. The difference for the alkane case $C_{10}H_{22}$ is accentuated because primitive pre-screening is likely to be more effective due to the spatial distribution of atoms of this molecule.

To understand the above Fock matrix construction timing results, for each of GTFock and NWChem, we measured the average time per process, T_{fock}, and the average computation-only time per process, T_{comp}. We assume that the average parallel overhead is $T_{ov} = T_{fock} - T_{comp}$. Figure 2 plots these quantities for different numbers of cores for our four test molecules.

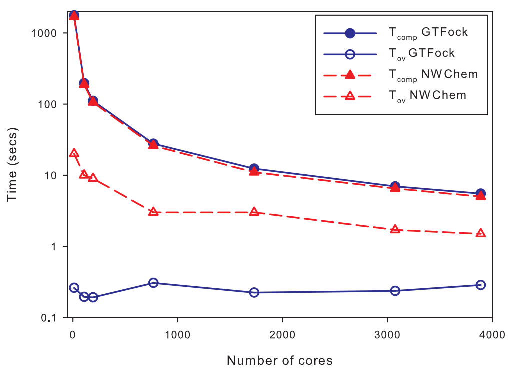
We note that, for all cases, the computation time for GTFock is comparable to that of NWChem, with computation in NWChem being slightly faster for reasons explained in the previous paragraph. However, in every case, the parallel overhead for GTFock is almost an order of magnitude lower than that for NWChem. For cases in Figures 2(a), (c), and (d), the overhead time for NWChem actually becomes comparable or greater than the average computation time for larger numbers of cores. This is due to the fact that there is relatively less work for these cases. For the alkane cases, there is less computational work because the molecules have a linear structure and there are many more shell quartets of integrals neglected due to screening. For the smaller graphene case, this is due to the fact that the amount of available computation is less. The increased proportion of parallel overhead is the reason for the poorer scalability of NWChem on these test cases, shown earlier in Table III and Table IV.

C. Analysis of Parallel Overhead

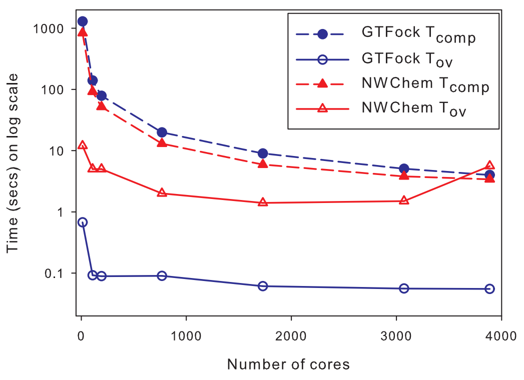
The parallel overhead time T_{ov}, illustrated in Figure 2, has three main sources: communication cost, load imbalance, and scheduler overhead from atomic accesses to task queues. We provide evidence to show the reduced communication cost of our algorithm versus that of NWChem. The cost of communication on a distributed system is composed of a latency term and a bandwidth term. The number of calls to communication functions in Global Arrays and the number of bytes transferred provide qualitative indicators of latency and bandwidth, respectively. We measured these quantities for NWChem and GTFock. These results are presented in Table VI and Table VII. We see that our implementation



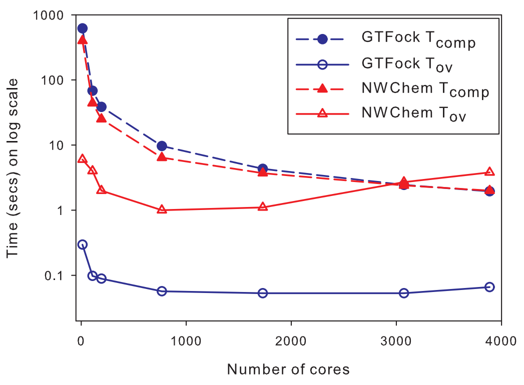
(a) $C_{96}H_{24}$



(b) $C_{150}H_{30}$



(c) $C_{144}H_{290}$



(d) $C_{100}H_{202}$

Figure 2: Comparison of average computation time T_{comp} and average parallel overhead time T_{ov} of Fock matrix construction for NWChem and GTFock. The computation times for NWChem and GTFock are comparable, but GTFock has much lower parallel overhead.

has lower volumes and numbers of calls for all the cases, indicating a lower communication cost, and explaining the reduced parallel overhead. It should be noted here that the volumes measured are total communication volumes, including local transfers. This was done in order to have a fair comparison between NWChem and GTFock since, as mentioned previously, the number MPI processes per core was different for each of them.

Scheduler overhead for NWChem can be inferred indirectly from the number of accesses to the task queue of its centralized dynamic scheduler. The number of such accesses for the case $C_{100}H_{202}$ with 3888 cores is 330091. Each of these operations must be atomic and it is expected that a serialization cost is incurred by them. In comparison, our work-stealing scheduler only needs the execution of 349 atomic operations on each of the task queues of the nodes. The serialization due to this is likely to be much less.

D. Load Balance

Our algorithm uses a work-stealing distributed dynamic scheduling algorithm to tackle the problem of load imbalance in Fock matrix construction. In this section we present experimental results that demonstrate the effectiveness of this approach to load balancing on our chosen test molecules.

Load balance can be expressed as the ratio of the longest time taken to complete the operations of Fock matrix construction for any process, to the average time. This is the ratio $l = T_{fock,max}/T_{fock,avg}$. A computation is perfectly load balanced if this ratio is exactly 1.

The ratios l for different numbers of processes, for the test molecules considered, are presented in Table VIII. The results indicate that in all the cases the computation is very well balanced, and that our load balancing approach is effective.

E. Performance of HF Iterations

The two major steps of the HF algorithm (Algorithm 1) are constructing the Fock matrix and computing the density matrix D. In this section we show that computing the density matrix comprises only a small portion of the running time of the HF algorithm for our largest test molecule.

Table VI: Average Global Arrays communication volume (MB) per MPI process for GTFock and NWChem.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	15.00	291.30	35.37	1020.79	40.61	457.16	84.08	1046.94
48	8.70	50.24	17.67	143.29	10.01	66.26	3.34	136.11
192	11.77	18.27	16.80	83.41	7.40	40.13	2.70	79.11
768	10.44	16.58	18.20	38.21	4.05	18.48	1.84	30.46
1728	7.63	11.36	12.87	22.03	3.56	14.87	3.90	17.95
3072	7.36	7.40	11.40	15.51	2.43	10.48	3.11	15.72
3888	6.24	8.23	9.94	16.37	2.38	8.26	2.78	14.44

Table VII: Average number of calls to Global Arrays communication functions per MPI process for GTFock and NWChem.

Cores	$C_{96}H_{24}$		$C_{150}H_{30}$		$C_{100}H_{202}$		$C_{144}H_{290}$	
	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem	GTFock	NWChem
12	11	3,204	11	10,758	11	9,590	11	19,898
48	33	678	71	1,401	28	1,379	29	2,587
192	59	610	81	822	28	955	33	1,510
768	65	602	103	840	28	988	38	1,023
1,728	127	711	123	758	32	1,386	35	950
3,072	129	533	148	637	30	1,253	32	1,189
3,888	147	1,091	170	1,008	30	989	31	1,357

Table VIII: Load balance ratio $l = T_{fock,max}/T_{fock,avg}$ for four test molecules. A value of 1.000 indicates perfect load balance.

Cores	$C_{96}H_{24}$	$C_{150}H_{30}$	$C_{100}H_{202}$	$C_{144}H_{290}$
12	1.000	1.000	1.000	1.000
108	1.021	1.011	1.015	1.023
192	1.031	1.019	1.024	1.022
768	1.026	1.031	1.021	1.027
1728	1.042	1.037	1.025	1.021
3072	1.039	1.035	1.032	1.023
3888	1.065	1.035	1.030	1.021

Table IX: Percentage of time taken by purification in GTFock for the $C_{150}H_{30}$ test case.

Cores	T_{fock}	T_{purf}	%
12	1765.921	19.534	1.09
108	196.954	4.984	2.47
192	111.016	3.982	3.46
768	28.027	1.913	6.39
1728	12.574	1.385	9.92
3072	7.207	1.203	14.30
3888	5.795	0.974	14.39

We implemented a "diagonalization-free" method for calculating the density matrix from the Fock matrix called *purification* [28]. This is an iterative method, in which each iteration involves two matrix multiply and trace operations. We implemented the SUMMA algorithm [29] for performing the matrix multiply operations in parallel.

Table IX shows results for the test case $C_{150}H_{30}$. The table shows the time for Fock matrix construction, T_{fock}, and for purification, T_{purf}, for the first HF iteration. Purification consists of 1 to 15% of the running time of an HF iteration for different numbers of cores. For this test case, purification converged in approximately 45 iterations, and it is expected that fewer iterations are required as the HF iteration itself converges. Note that no data redistribution of the distributed F and D matrices was needed after the Fock matrix construction step and before the application of purification. The distribution of F and D is exactly the distribution needed for the SUMMA algorithm.

V. Conclusions

This paper presented a new scalable algorithm for Fock matrix construction for the HF algorithm. We addressed two issues: load balance and the reduction of communication costs. Load balance was addressed by using fine-grained tasks, an initial static task partitioning, and a work-stealing dynamic scheduler. The initial static task partitioning, augmented by a reordering scheme, promoted data reuse and reduced communication costs. Our algorithm has measurably lower parallel overhead than the algorithm used in NWChem for moderately-sized problems chosen to accentuate scalability issues.

We expect that the technology for computing ERIs will improve, and efficient full-fledged implementations of ERI algorithms on GPUs will reduce computation time to a fraction of its present cost. This will, in turn, increase the significance of new algorithms such as ours that reduce parallel overhead.

Several avenues are open for future research. The identification of improved reordering schemes, and the use of new "smart" distributed dynamic scheduling algorithms would lead to improved performance.

Acknowledgments

This research was supported by the NSF under grant ACI-1147843. NSF XSEDE computer resources were provided under a S2I2 conceptualization grant, ACI-121682. The authors gratefully acknowledge the help of Prof. D. Sherrill of the School of Chemistry and Biochemistry at the Georgia Institute of Technology.

References

[1] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover, 1989.

[2] R. J. Bartlett, "Many-body perturbation theory and coupled cluster theory for electron correlation in molecules," *Annual Review of Physical Chemistry*, vol. 32, no. 1, pp. 359–401, 1981.

[3] J. A. Pople, P. M. Gill, and B. G. Johnson, "Kohn-Sham density-functional theory within a finite basis set," *Chemical Physics Letters*, vol. 199, no. 6, pp. 557–560, 1992.

[4] I. S. Ufimtsev and T. J. Martinez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[5] N. Luehr, I. S. Ufimtsev, and T. J. Martínez, "Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011.

[6] K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2008.

[7] Y. Miao and K. M. Merz, "Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations," *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, 2013.

[8] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, "Uncontracted Rys quadrature implementation of up to g functions on graphical processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.

[9] K. A. Wilkinson, P. Sherwood, M. F. Guest, and K. J. Naidoo, "Acceleration of the GAMESS-UK electronic structure package on graphical processing units," *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2313–2318, 2011.

[10] T. Ramdas, G. K. Egan, D. Abramson, and K. K. Baldridge, "On ERI sorting for SIMD execution of large-scale Hartree-Fock SCF," *Computer Physics Communications*, vol. 178, no. 11, pp. 817–834, 2008.

[11] ——, "ERI sorting for emerging processor architectures," *Computer Physics Communications*, vol. 180, no. 8, pp. 1221–1229, 2009.

[12] R. J. Harrison, M. F. Guest, R. A. Kendall, D. E. Bernholdt, A. T. Wong, M. Stave, J. L. Anchell, A. C. Hess, R. Littlefield, G. I. Fann *et al.*, "High performance computational chemistry: II. a scalable SCF program," *J. Comp. Chem*, vol. 17, p. 124, 1993.

[13] I. T. Foster, J. L. Tilson, A. F. Wagner, R. L. Shepard, R. J. Harrison, R. A. Kendall, and R. J. Littlefield, "Toward high-performance computational chemistry: I. Scalable Fock matrix construction algorithms," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 109–123, 1996.

[14] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 53.

[15] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[16] E. Aprà, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid water: obtaining the right answer for the right reasons," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 66.

[17] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy *et al.*, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[18] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[19] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su *et al.*, "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[20] V. Lotrich, N. Flocke, M. Ponton, A. Yau, A. Perera, E. Deumens, and R. Bartlett, "Parallel implementation of electronic structure energy, gradient, and hessian calculations," *The Journal of Chemical Physics*, vol. 128, p. 194104, 2008.

[21] C. L. Janssen and I. M. Nielsen, *Parallel Computing in Quantum Chemistry*. CRC Press, 2008.

[22] D. L. Strout and G. E. Scuseria, "A quantitative study of the scaling properties of the Hartree-Fock method," *The Journal of Chemical Physics*, vol. 102, p. 8448, 1995.

[23] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[24] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[25] http://http://www.tacc.utexas.edu/resources/hpcsystems/.

[26] N. Flocke and V. Lotrich, “Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations,” *Journal of Computational Chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.

[27] T. H. Dunning Jr, “Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen,” *The Journal of Chemical Physics*, vol. 90, p. 1007, 1989.

[28] A. H. Palser and D. E. Manolopoulos, “Canonical purification of the density matrix in electronic-structure theory,” *Physical Review B*, vol. 58, pp. 75 524–12 711, 1998.

[29] R. A. Van De Geijn and J. Watts, “SUMMA: Scalable universal matrix multiplication algorithm,” *Concurrency—Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.