High Performance Computational Chemistry: (I) Scalable Fock Matrix Construction Algorithms

Ian T. Foster, Jeffrey L. Tilson, Albert F. Wagner, Ron Shepard
Argonne National Laboratory
Argonne, IL 60439

Robert J. Harrison, Rick A. Kendall, Rik J. Littlefield,
Molecular Science Research Center
Pacific Northwest Laboratory
Richland,WA 99352

Abstract

Several parallel algorithms for Fock matrix construction are described. The algorithms calculate only the unique integrals, distribute the Fock and density matrices over the processors of a massively parallel computer, use blocking techniques to construct the distributed data structures, and use clustering techniques on each processor to maximize data reuse. Algorithms based on both square and row blocked distributions of the Fock and density matrices are described and evaluated. Variants of the algorithms are discussed that use either triple-sort or canonical ordering of integrals, and dynamic or static task clustering schemes. The algorithms are shown to adapt to screening, with communication volume scaling down with computation costs. Modeling techniques are used to characterize algorithm performance. Given the characteristics of existing massively parallel computers, all the algorithms are shown to be highly efficient on problems of moderate size. The algorithms using the row blocked data distribution are the most efficient.

1 Introduction

The high computational power and large aggregate memory of massively parallel processing (MPP) supercomputers gives these machines the potential to solve Grand Challenge-class problems in computational chemistry. In this and a companion paper [1], we report

our initial efforts to develop effective *ab initio* electronic structure codes for MPP computers that are capable of solving problems with $\mathcal{O}(10^{2-3})$ atoms and $\mathcal{O}(10^{3-4})$ basis functions. Problems of this scale almost automatically imply that all matrices be distributed over the processors. The present paper reviews a number of strategies for coding relatively simple electronic structure methods when using distributed data structures. The companion paper describes a specific implementation and presents initial performance characteristics of electronic structure applications to problems of up to 100 atoms and 1000 basis functions.

The architecture of MPP computers is very different from that of vector supercomputers. A typical MPP computer consists of a collection of processors, each with its own memory and connected via a high performance network. When designing algorithms for these computers, important issues include avoiding replicated computation (*computational efficiency*), distributing data structures so as to avoid wasting memory (*data distribution*), distributing computation to processors so as to avoid idle time when one processor is busy and others are not (*load balance*), and minimizing time spent sending and receiving messages (*communication efficiency*). A metric that integrates these different criteria is *scalability*: the extent to which an algorithm is able to solve larger problems as the number of processors is increased.

The complex architecture of MPP computers makes intuitive notions of performance unreliable. Hence, a sound methodology when developing parallel algorithms is to begin by examining algorithmic alternatives at a theoretical level. Only once scalability has been established should effort be devoted to implementations on parallel computers. In this paper, we apply this methodology to the direct closed-shell self-consistent field (SCF) method [2, 3]. See [4] (and references therein) for a recent review of current parallel SCF development. SCF is an important method in its own right and, in addition, is typical of other more sophisticated methods in its use of large data structures and irregular data access patterns. We develop a family of algorithms for the Fock matrix construction component of the SCF method, analyze the scalability of these algorithms, and make recommendations as to which algorithm to incorporate in a parallel implementation.

The SCF method obtains the energy and wavefunction of a molecular system by iterating over two basic steps until self-consistency is obtained. First, a two-dimensional Fock matrix F is constructed from the current estimate of the wavefunction. Second, F is diagonalized to obtain an improved estimate of the wavefunction. The second step is computationally trivial on sequential computers but can become a rate limiting step on large numbers of processors [6]. In future work, we will explore an alternative scheme proposed by Shepard [7] that avoids the need for diagonalization.

We focus on the Fock matrix construction problem in this paper. The Fock matrix F in the atomic orbital (AO) basis is defined as

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

where N is the number of basis functions, h is the one-electron Hamiltonian, D is the one-particle density matrix, and $(ij|kl)$ represents a two-electron integral. All quantities are assumed hereafter to be real. The calculation of the two-electron integrals is the most expensive component of this computation. It might appear from Eqn. 1 that N^4 integrals

must be evaluated. However, D and F are symmetric and for any (i,j,k,l) the following integrals are equivalent:

$$(ij|kl) = (ji|kl) = (ij|lk) = (ji|lk) = (kl|ij) = (kl|ji) = (lk|ij) = (lk|ji). \qquad (2)$$

Once $(ij|kl)$ is computed, the related elements of F (F_{ij}, F_{ik}, F_{il}, F_{jk}, F_{jl}, and F_{kl}) can be updated with the product of this integral and the appropriate element of D. Hence, the total number of integrals to be computed is only

$$N_{\text{intg}} \quad = \quad \left(\begin{array}{c} \left(\frac{N+1}{2}+1\right) \\ 2 \end{array} \right) \approx \frac{N^4}{8} \qquad (3)$$

In order to exploit this symmetry, each integral calculation requires up to six D-elements and contributes to at most six F-elements.

The issue of *screening* must also be addressed. For large molecules, most integrals are so small that their contribution to F is negligible. For large molecules, screening can reduce the number of contributing integrals from $\mathcal{O}(N^4)$ to close to $\mathcal{O}(N^2)$.

Early parallel SCF programs either replicated the D and F matrices in each processor of a parallel computer or had one processor maintain the data and control which processor computes an integral batch. In all cases a resulting F matrix would reside on a single processor for analysis [8-17]. This approach simplified implementation and achieved high performance. However, the replicated data restricted scalability: the maximum problem size that could be solved was limited by the amount of memory on a single processor. For example, Feyereisen and Kendall's parallel DISCO code [15] is limited to approximately 400 basis functions (without symmetry) on the 512-processor Intel Touchstone Delta computer, which has 16 MB of memory per processor. Nevertheless, these studies provided much useful information on the distribution of computational tasks, load balancing and task scheduling, etc.

A scalable parallel Fock matrix construction algorithm must distribute the D and F matrices over available processors, so that the maximum problem size is limited only by the aggregate memory available on the MPP computer. In Colvin *et al.*'s systolic algorithm [18], Fock and density submatrices are circulated among processors. However, this approach requires the computation of $3N^4/8$ integrals and suffers from an overly synchronous computational model. Furlani and King [19] describe an algorithm that avoids these deficiencies. Their algorithm uses several of the techniques discussed in this paper, including a two-dimensional blocked distribution of Fock and density matrices, the use of static and dynamic scheduling to balance the computational load, the reuse of local data, and agglomeration of integral computations into larger tasks. However, they do not analyze their algorithm's parallel scalability.

The algorithms presented in this paper distribute the principal data structures, avoid redundant integral computation, provide a framework for addressing the load balancing problem, and can be adapted to highly screened or unscreened calculations. In addition, analysis shows that communication costs are significantly less than computation costs on most reasonable computer systems. Hence, the algorithms are expected to be efficient and scalable. The results of this analysis are confirmed in the companion paper [1].

The rest of the paper is organized as follows. In Section 2, we describe the basic features of the algorithm in the absence of screening. In Section 3, we describe variations

that can make the basic algorithm more versatile and efficient. In Section 4, we discuss the ramifications of screening. In Section 5, we present a performance analysis of selected algorithm variants. In Section 6, we summarize.

2 Blocked Fock Matrix Construction Algorithms

Our Fock matrix construction algorithm is computationally efficient: in the absence of screening it performs only the essential $\mathcal{O}(N^4/8)$ integral evaluations. Hence, in evaluating its performance we shall focus on its communication requirements and load balance. On many parallel computers, the cost of a communication can be modeled with reasonable accuracy as a function of a startup cost, t_0, and a per-word cost, t_1. This measure neglects both network and node contention. If we characterize the communication requirements of an algorithm in terms of the total volume of data moved between processors, V, and the number of messages sent, M, then the total communications time $T_{communicate}$ is:

$$T_{communicate} = Mt_0 + Vt_1. \tag{4}$$

To illustrate, consider a simple parallel algorithm where there is a random distribution among processors of both integral evaluations and F and D matrix elements. Each integral evaluation requires six D elements and computes contributions to six F elements. For large P, where P is the number of processors, an integral and its data elements will almost always be located on different processors, so the total communication requirements will be

$$M = V = \frac{3}{2}N^4. \tag{5}$$

On most parallel computers, the cost of sending a message is a substantial fraction of the time required for computing an integral. As this algorithm generates twelve messages per integral, it is unlikely to be efficient. Our goal in designing efficient parallel algorithms is to reduce both the number of messages sent and the total volume of data communicated.

2.1 A Blocked Algorithm

A commonly-used technique for reducing communication requirements in parallel algorithms is to *block* computations into larger *tasks*. This technique can reduce communication requirements if computations in a task read and write the same data, then both the number of messages and the total volume of data communicated can be reduced.

We can incorporate blocking into the Fock matrix construction algorithm by redefining the (i, j, k, l) in Eqn. 1 to index symmetry-distinct *atomic center* integrals, where each integral now represents a batch of *basis function* integrals and can be designated a task. We could also interpret these indices as referring to shells, molecular fragments, etc.

We will show below that the data accessed by a task shows considerable locality, and that this locality can be exploited in a number of ways to reduce communication requirements. For now, we assume simply that the D and F matrices are distributed using a two-dimensional blockwise distribution of the sort illustrated in Fig. 1. When the

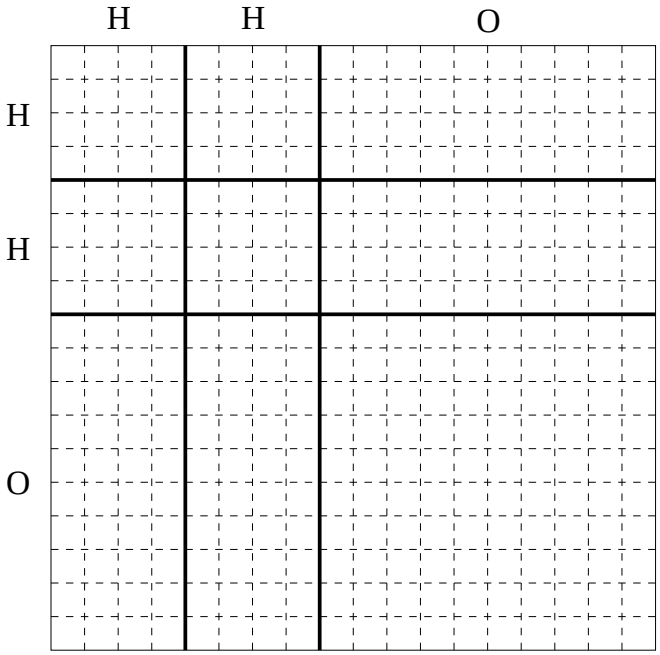

Figure 1: The density matrix for H_2O, assuming a basis set that allocates 4 basis functions to each H and 10 to O. Hence, $N = 4 + 4 + 10 = 18$, and $I_c = N/\text{n(atoms)}=6$. The solid lines delineate submatrices Dij; each submatrix is located on a single processor and hence can be communicated in a single message.

indices in Eqn. 1 are interpreted as referring to atomic centers, each Dij in that equation represents a submatrix, which in this distribution is located on a single processor and can be communicated in a single message. Each submatrix will contain approximately $I_c \times I_c$ elements, where I_c is the number of basis functions divided by the number of atoms. However, as the number of basis functions varies with the type of atom, submatrices need not be square.

We can now determine the total communications requirements for this blocked algorithm. The total number of messages per task is still 12 as in Eqn. 5 but the total number of tasks becomes

$$\begin{aligned} Tasks &\approx \sum_{i=1,I_c}^{N} \sum_{j=1,I_c}^{i} \sum_{k=1,I_c}^{i} \sum_{l=1,I_c}^{k} 1 \\ &\approx \frac{1}{8}\left(\frac{N}{I_c}\right)^4. \end{aligned}$$

The use of I_c signifies a stride on the indices. The total number of messages becomes

$$M_{\text{blocked}} = 12 \times Tasks \approx \frac{3}{2}\left(\frac{N}{I_c}\right)^4. \tag{6}$$

As each message transfers a submatrix of $I_c \times I_c$ elements. The total volume transferred is:

$$V_{\text{blocked}} \approx \frac{3N^4}{2(I_c)^2}. \tag{7}$$

We see that blocking reduces message counts by a factor of $(I_c)^4$ and volume by $(I_c)^2$. When blocking by atoms, I_c is typically 10 or greater, so the reduction in communication requirements achieved with this technique is substantial.

We now consider more sophisticated blocking strategies. First, we must consider the order in which the unique integrals for the construction of F are evaluated. We shall discuss parallel algorithms based on both a *canonical* ordering (specified in Fig. 2) or a "*triple-sort*" ordering (Fig. 3). In the triple-sort ordering, the `compute(`i`,`j`,`k`,`l`)` operation computes up to three batches of integrals if the index permutations lead to symmetry-distinct integrals. In the canonical ordering, this operation computes exactly one batch of integrals. In a sequential computer environment, the two orderings differ only slightly in their performance. We shall show that in a parallel environment, their communication requirements and screening characteristics can be quite different.

For simplicity, the indices in Figs 2 and 3 refer to basis functions, and a fixed stride of I_c is used. However, it is straightforward to modify the figures to index shells, atoms, etc., directly, with unit stride. The screening tests in each figure will be discussed in Section 4. In this and the succeeding section, we assume that all index values survive screening.

Consider the impact of choosing the `innerloop(`i`,`j`,`k`)` operation of these figures as a task. If m is set to i for canonical and j for triple-sort, the number of `innerloop(`i`,`j`,`k`)` tasks is:

$$\sum_{i=1,I_c}^{N} \sum_{j=1,I_c}^{i} \sum_{k=1,I_c}^{m} 1 \approx \frac{N^3}{6(I_c)^3} \qquad triple-sort \tag{8}$$

$$\approx \frac{N^3}{3(I_c)^3} \qquad canonical \tag{9}$$

In both orderings, the increased granularity has decreased the number of tasks by $\mathcal{O}(N/I_c)$. Canonical ordering has twice as many tasks as triple-sort.

Each `compute(`i`,`j`,`k`,`l`)` operation reads and writes data at locations (ij), (ik), (il), (jk), (jl), and (kl). Hence, each `innerloop(`i`,`j`,`k`)` task accesses the following elements of D and F. (Due to blocking, each of these "elements" is a submatrix of size $I_c \times I_c$.)

1. $(i\ j)$ $\{\ [\ i,j\ ]\ \}$
2. $(i\ k)$ $\{\ [\ i,k\ ]\ \}$
3. $(j\ k)$ $\{\ [\ \texttt{max}(j,k),\texttt{min}(j,k)\ ]\ \}$
4. $(i\ l)$ $\{\ [\ i,l\ ]\ \mid\ 1 \le l \le \texttt{ltop}\ \}$
5. $(k\ l)$ $\{\ [\ i,l\ ]\ \mid\ 1 \le l \le \texttt{ltop}\ \}$
6. $(j\ l)$ $\{\ [\ k,l\ ]\ \mid\ 1 \le l \le \texttt{ltop}\ \}$

In this table, `ltop` is k for triple-sort order and the `lhi` of Fig. 2 for canonical. This list exploits the symmetry of F and D in both orderings. The data for each task comes

```
fock_build
DO i = 1, N, I_c
   DO j = 1, i, I_c
      IF (i,j pair survive screening) THEN
         DO k = 1, i, I_c
            CALL innerloop(i,j,k)
         ENDDO
      ENDIF
   ENDDO
ENDDO

innerloop(i,j,k)
IF (k.EQ.i) lhi = j
IF (k.NE.i) lhi = k
DO l = 1, lhi, I_c
   IF (k,l pair survive screening) THEN
      CALL compute(i,j,k,l)
   ENDIF
ENDDO

compute(i,j,k,l)
DO FOR STRIDE OF i, j, k, l
   EVALUATE I_intg = (ij|kl)
   F_ij =  F_ij + D_kl I_intg
   F_kl =  F_kl + D_ij I_intg
   F_ik =  F_ik - 1/2 D_jl I_intg
   F_il =  F_il - 1/2 D_jk I_intg
   F_jl =  F_jl - 1/2 D_ik I_intg
   F_jk =  F_jk - 1/2 D_il I_intg
ENDDO
```

Figure 2: Basic logic for Fock matrix construction: canonical order

```
fock_build
DO i = 1, N, I_c
   DO j = 1, i, I_c
      IF (i,j pair survive screening) THEN
         DO k = 1, j, I_c
            CALL innerloop(i,j,k)
         ENDDO
      ENDIF ENDDO
ENDDO

innerloop(i,j,k)
DO l = 1, k, I_c
   CALL compute(i,j,k,l)
ENDDO

compute(i,j,k,l)
DO FOR STRIDE OF i, j, k, l
   IF(k,l pair survive screening) EVALUATE I_1 = (ij|kl)
   IF([i,k and j,l pairs survive screening].AND.
      [distinct]) EVALUATE I_2 = (ik|jl)
   IF([i,l and j,k pairs survive screening].AND.
      [distinct]) EVALUATE I_3 = (il|jk)
   DO n = 1,3 ( Distinct In only )
      I_intg = I_n
      F_ij =  F_ij + D_kl I_intg
      F_kl =  F_kl + D_ij I_intg
      F_ik =  F_ik - 1/2 D_jl I_intg
      F_il =  F_il - 1/2 D_jk I_intg
      F_jl =  F_jl - 1/2 D_ik I_intg
      F_jk =  F_jk - 1/2 D_il I_intg
   ENDDO
ENDDO
```

Figure 3: Basic logic for Fock matrix construction: triple-sort order

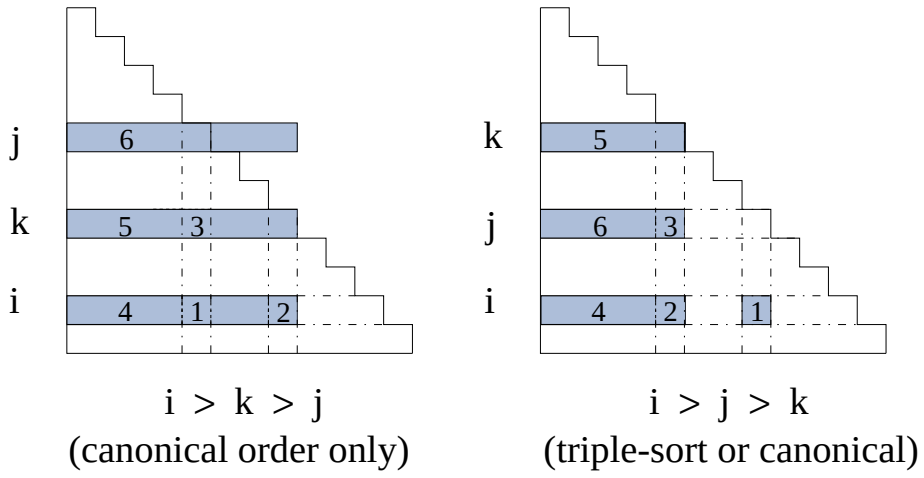

Figure 4: Data requirements for an (ijk) task, in the absence of screening. The numbers refer to the data requirements listed in the text.

from the ith, jth, and kth rows of D and F. Furthermore, the first 5 elements all come from the lower triangle of D and F. The sixth element also comes from the lower triangle unless canonical order is used and $i > k > j$ (a situation that occurs approximately half the time in canonical order). These data requirements are illustrated in Fig. 4 where i, j, and k should be thought of as indexing blocks of I_c rows.

We see that the data accessed by the integrals in an `innerloop(`i`,`j`,`k`)` task do indeed show considerable locality. We can exploit this locality by distributing the D matrix as follows. We create N/I_c data sets, D_1, ..., D_{N/I_c}. Each data set comprises an entire row of submatrix blocks (in the canonical algorithm) or partial row to the diagonal (in the triple-sort algorithm). These data sets are then distributed over the available processors. The F matrix is distributed in the same manner.

With this data distribution strategy, the communication requirements of an `innerloop` task can be satisfied with just six messages: three before the task is executed (one from each of the processors holding D_i, D_j, and D_k) and three afterwards (to the same processors, for F_i, F_j, and F_k). Notice that for efficiency, only those elements of D_i, etc., that are required by a particular task should be communicated. (As illustrated in Fig. 4, each task requires at most $(I_c \times I_c)$ $3k+1$ elements of D.) The communication requirements are then as follows.

$$M_{\text{row\_blocked\_triple\_sort}} \approx \frac{N^3}{(I_c)^3} \tag{10}$$

$$V_{\text{row\_blocked\_triple\_sort}} \approx \frac{1}{4}\frac{N^4}{(I_c)^2} \tag{11}$$

$$M_{\text{row\_blocked\_canonical}} \approx 2\frac{N^3}{(I_c)^3} \tag{12}$$

$$V_{\text{row\_blocked\_canonical}} \approx \frac{3}{4}\frac{N^4}{(I_c)^2} \tag{13}$$

In summary, the blocking of integrals into `innerloop(`i`,`j`,`k`)` tasks reduces M by $\mathcal{O}(N/I_c)$ and V by a constant factor relative to the blocked algorithm. For the row-blocked algorithm, triple-sort order has better performance than canonical: its M and V are less by factors of 2 and 3, respectively. In addition, triple-sort stores only the lower triangle of the D and F matrices: half as much data as canonical sort.

2.2 Symmetrization

Another issue influencing the relative performance of the canonical and triple sort orderings is *symmetrization*. As illustrated in Fig. 4, the canonical ordering produces an F that must be symmetrized. The actual arithmetic operations involved in the symmetrization, namely summing the symmetrically related elements of F, will normally be insignificant. However, the operation has the same communication requirements as a parallel matrix transpose [21, 22]. The usual algorithm requires that each processor exchange data with every other processor, for a total of P^2 communications on a computer with P processors. Assuming $\mathcal{O}(N)$ processors, the total communication costs are $\mathcal{O}(N^2)$ data in $\mathcal{O}(N^2)$ messages. In the absence of screening, these costs are at least $\mathcal{O}(N)$ less than those associated with the construction of F. However, they may become significant if screening reduces communication costs during F construction to $\mathcal{O}(N^2)$.

2.3 Load Balance

Tasks can vary significantly in their computational cost, for three reasons. First, the computational cost of a single integral can vary greatly. (For simple hydrocarbons, the time to calculate $(ij|kl)$ integrals can vary by orders of magnitude depending on the number of basis functions indexed by i, j, k, and l.) Second, as discussed in Section 4 below, screening can alter the number of integrals actually computed. Third, for the row-blocked algorithm, the number of integrals in a task depends on k.

A consequence of this variation in task cost is that an allocation strategy that places an equal number of tasks on each processor may suffer from load imbalances. One solution to this problem is to use a centralized scheduler to allocate tasks to processors in a demand-driven manner [15, 19]. Scheduler-based techniques can achieve excellent load balance, but increase communication requirements and are not truly scalable. Alternatively, a random allocation strategy can be used. This relies on the law of large numbers to balance the computational load. If $P \leq N$, each processor will have on average $\mathcal{O}(N^2)$ tasks without screening and, as will be shown later, $\mathcal{O}(N)$ tasks with screening. This suggests that a random allocation approach will be load balanced. This should also be true of the blocked algorithm where the random allocation would be at the head of the l loop in the `innerloop` of Figs 2 and 3. Hybrid schemes that use both a scheduler and random mapping are also possible [19].

2.4 Blocked Algorithm Summary

Blocking is equivalent to the use of a non-unit stride I_c in the nested loops executed when constructing the Fock matrix. We have described and analyzed the performance, in the absence of screening, of both simple blocked and row blocked algorithms, based on both triple-sort and canonical orderings of integrals. All algorithms evaluate the minimum number of integrals and support a full distribution of the D and F matrices. Blocking is shown to reduce communication requirements by a factor of between $(I_c)^2$ and $(I_c)^4$, depending on the algorithm used. Row blocking is more efficient than simple blocking. The triple-sort algorithm is superior to the canonical algorithm: it has lower communications costs, uses one half as much memory, and does not require the $M = V = \mathcal{O}(N^2)$ F matrix symmetrization.

3 Clustering

The analysis in the previous section showed that the blocking of integrals into tasks improves parallel performance by reducing communication requirements. Hence, we might wonder whether it is beneficial to group tasks into even larger collections. In this section, we describe two techniques for organizing tasks into larger collections that we call *clusters*, and show that both techniques can reduce communication costs in certain circumstances. The first technique supports a dynamic mapping strategy while the second defines a static mapping. These techniques are described in the context of the row-blocked algorithm and distribution scheme described in Section 2

3.1 Dynamic Clustering

In the first clustering algorithm that we consider, each processor executes the logic specified in Fig. 5. The `assigned_to_me` function determines whether a particular (i, j, k) task is intended for that processor. As discussed in the preceding section, this could be implemented as a call to a scheduler or as some "random" function.

The functions `get` and `put` represent the communication operations required to fetch D_k and store F_k, respectively. The function `conditional_put` performs a communication only if the index of the row to which it is applied has changed since the last iteration. The `conditional_get` performs a communication if the requested elements of D are not already locally stored. Thus a change in the row index since the last iteration will certainly generate a communication. However, a change in the requested column indeces for a fixed row may or may not generate a communication. The `conditional_gets` and `_puts` allow elements of data sets D_i, D_j, F_i, and F_j to be reused or *cached*, thereby reducing total communication requirements.

We now analyze the communication requirements of this clustered algorithm. We consider the triple-sort ordering; the canonical ordering is similar. We consider the communication requirements of the D_k/F_k, D_j/F_j, and D_i/F_i pairs in turn.

(D_k/F_k): The contributions to both M and V are as in the row-blocked algorithm: that is, 1/3 the values given in Eqns 10 and 11.

```
DO i = 1, N, I_c
   DO j = 1, i, I_c
      IF (i,j pair survive screening) THEN
         DO k = 1, m, I_c
            IF (assigned_to_me(ijk)) THEN
               CALL conditional_get(D_i)
               CALL conditional_put(F_i)
               CALL conditional_get(D_j)
               CALL conditional_put(F_j)
               CALL get D_k
               CALL innerloop(i,j,k)
               CALL put(F_k)
            ENDIF
         ENDDO
      ENDIF ENDDO
ENDDO
```

Figure 5: Clustered algorithm [$m = j$ (triple-sort) or i (canonical)]

(D_j/F_j): The number of messages generated for D_j is as in the row-blocked algorithm: i.e., one sixth of $M_{\text{row\_blocked}}$. In contrast, the number of messages for F_j is bounded from above by $\texttt{min}(P,j/I_c)$ for each (i,j) pair. This is because the F_j data transfer occurs after all tasks in the pair are complete; hence, each of P processors generates at most one message. If there are fewer tasks than processors (i.e., if $P > j/I_c$), then at most j/I_c messages are generated. The number of messages can be less than this upper bound if tasks are not maximally dispersed over processors. (In the unlikely event that all of these tasks were assigned to one processor, then there would be only one message for F_j.)

D_j and F_j make identical contributions to V. From Fig. 4, we see that the D_j elements read by tasks in a series with fixed (i,j) but increasing k are precisely those elements read when processing the highest k value. Assuming that the highest k values are distributed evenly among processors, the total number of D_j elements read by all processors will be the size of a single element times the sum of the P (or j/I_c, if $P > j/I_c$) highest k values. With blocking, a single element has size $(I_c)^2$, and we sum the $\texttt{min}(P,j/I_c)$ highest values of k/I_c. This is an upper bound: fewer elements will be read if the highest k values are not evenly distributed over processors. A similar analysis applies to the F_j elements.

(D_i/F_i): Tasks with a fixed i can assume the same value of k multiple times. With triple-sort ordering, there are i/I_c values of k that begin a stride. Each of these values will, because of the loop over j, occur $(i-k)/I_c$ times, making the total number of (j,k)-strided pairs approximately $(i/I_c)^2/2$. For D_i, for each allowable value of k, the number of messages must be bounded from above by $\texttt{min}(P,(i-k)/I_c)$. For F_i the number of messages is, by analogy with F_j, bounded from above by $\texttt{min}(P,(i/I_c)^2/2)$. The message volume for D_i is the same as that for F_i. By analogy with D_j, the V contribution of D_i is

bounded from above by $(I_c)^2$ elements times the $\texttt{min}(P,(i/I_c)^2/2)$ highest values of k/I_c (including the multiple times a k/I_c value occurs).

From these considerations, we obtain the following expressions for the *clustered* algorithm for triple-sort ordering:

$$M_{\text{clustered}} = \frac{1}{2}M_{\text{row\_blocked}} + M_{Fj} + M_{Fi} + M_{Di} \tag{14}$$

$$V_{\text{clustered}} = \frac{1}{3}V_{\text{row\_blocked}} + V_j + V_i \tag{15}$$

where

$$M_{Fj} = \sum_{i=1,I_c}^{N} \sum_{j=1,I_c}^{i} \texttt{min}(P, j/I_c)$$

$$M_{Fi} = \sum_{i=1,I_c}^{N} \texttt{min}(P, (i/I_c)^2/2)$$

$$M_{Di} = \sum_{i=1,I_c}^{N} \sum_{k=1,I_c}^{i} \texttt{min}(P, (i-k)/I_c)$$

$$V_j = 2\sum_{i=1,I_c}^{N} \sum_{j=1,I_c}^{i} \sum_{k=\texttt{max}(1,j-I_cP),I_c}^{j} (I_c)^2(k/I_c)$$

$$V_i = 2\sum_{i=1,I_c}^{N} \sum_{k=\texttt{max}(1,i-I_c(2P)^{1/2}),I_c}^{i} [(i-k)/I_c](I_c)^2(k/I_c)$$

The lower limits on the j and k summation in the expressions for V_j and V_i correspond to the $\texttt{min}(P,j/I_c)$ or $\texttt{min}(P,(i/I_c)^2/2)$ highest values of k/I_c discussed above.

These expressions demonstrate the value of clustering, particularly for smaller P. Communication costs range from approximately 1/3 those of the row-blocked algorithm (for small P) to approximately the same (for large P).

3.2 Static Clustering

The clustering scheme presented above exploits reuse within clusters of $\mathcal{O}((N/I_c)^2/P)$ tasks. We now present an alternative technique that constructs much larger clusters containing $\mathcal{O}((N/I_c)^2)$ tasks. This provides additional opportunities for reuse. In an attempt to ensure load balance, these clusters are constructed so that in the absence of screening, each contains the same number of integrals.

As noted previously, one reason why task costs vary is the length of the l loop. In the absence of screening, this length is determined entirely by the i, j, and k indices. Hence, it is possible to cluster tasks to generate *supertasks* containing a constant number of l values (integrals). Supertasks are then mapped to processors either randomly or using a scheduler. If supertasks have the property that two of the three task indices remained the

```
Repeat for x = p and x = N − p + 1:
   i = x
   DO j = 2, i, 2
      DO k = 1, i
         CALL innerloop(i,j,k)
      ENDDO
   ENDDO
   k = x
   DO i = x, N
      DO j = 1, i, 2
         CALL innerloop(i,j,k)
      ENDDO
   ENDDO
```

Figure 6: Algorithm for Constructing Supertask Number p

same while the third ranges, on average, over a large number of values, then the number of messages per task for unit strides approaches two. This corresponds to the small P limit for the clustered algorithm and is the minimum number possible without data replication.

We define one possible supertasking strategy. For convenience this strategy will be described in terms of a unit stride for canonical ordering. However, it can be adapted for a non-unit stride and triple-sort ordering.

Consider $N/2$ supertasks, each containing approximately $N^3/4$ integrals. The $N/2$ supertasks are numbered $1..N/2$ and indexed by p. The algorithm for supertask p is given in Fig. 6 for canonical ordering and for a common unit stride. It is easy to show that this clustering algorithm, designated *static_cluster*, does indeed yield supertasks containing the same number of integrals. From Fig. 6, when $x = p$, we have

$$\begin{aligned}
\sum_{j=2 \text{ by } 2}^{p} \sum_{k=1}^{p} k \;+\; \sum_{i=p}^{N} \sum_{j=1 \text{ by } 2}^{i} p \;&\approx\; \frac{p}{2}\frac{p(p+1)}{2} \;+\; p\sum_{i=p}^{N}\frac{i}{2} \\
&=\; \frac{p^2(p+1)}{4} \;+\; \frac{p}{2}\sum_{i=1}^{N-p}(i+p) \\
&=\; \frac{p^2(p+1)}{4} + \frac{p}{2}\left(\frac{(N-p)(N-p+1)}{2} + p(N-p)\right) \\
&=\; \frac{pN(N+1)}{4} \\
&\approx\; \frac{pN^2}{4}
\end{aligned}$$

As supertask p comprises integrals for $x = p$ and $x = N - p+1$, the total number of

integrals is approximately

$$\frac{pN^2}{4} + \frac{(N-p)N^2}{4} = \frac{N^3}{4},$$

which is a constant for fixed N. Note that the number of tasks per supertask is *not* a constant; this is approximately

$$\frac{3N^2 + 2p(p-N)}{4},$$

which, as p is in the range $(0, N/2)$, ranges over $(5/2N^2, 3N^2)$ for large N.

Each supertask has four double-loop structures. For each of the four double loops, one of the indices of all the tasks to be included is fixed. A second index varies slowly and the last index varies over, on average, a large range of consecutive values. It is easy to show that, for large N, communication requirements are related as follows. This relationship applies for all values of P up to $N/2$, the total number of supertasks.

$$\frac{M_{\text{static\_cluster}}}{M_{\text{row\_blocked}}} = \frac{V_{\text{static\_cluster}}}{V_{\text{row\_blocked}}} = \frac{1}{3} \tag{16}$$

3.3 Clustering Summary

The clustering of task collections to enhance reuse of D and F matrix values is a general technique that can enhance the performance of a parallel Fock matrix construction algorithm by reducing communication costs. Both the static and dynamic clustering schemes presented in this section have been shown to scale down communications requirements by as much as 2/3. In the dynamic scheme, the reduction in communication costs drops off as P increases; in the static scheme, the reduction of 2/3 is maintained even for large P. However, we shall see in the next section that static clustering is not effective in the presence of screening.

Notice that while we have focused here on the row-blocked algorithm, the blocked algorithm can also be adapted to clustering by the use of `conditional_get` and `_put` statements as in Fig. 5. This strategy is employed in the code described in the companion paper, which uses a clustered version of the blocked algorithm with an atom-indexed blocking of data.

4 Screening

Each integral computed when constructing the Fock matrix represents the coulombic interaction of two pairs of overlapping basis functions. If either a pair of basis functions does not overlap significantly or if the interaction of the two pairs is negligible then the corresponding integral will not contribute significantly to the final F matrix and need not be computed. Screening may be accomplished by the use of a tolerance limit on the integrals, specified in Figs 2 and 3. In large molecules, this screening criterion can reduce the cost of constructing F from a nominal $\mathcal{O}(N^4)$ integrals to, in the limit, $\mathcal{O}(N^2)$ [3, 5].

The effect of screening on communication requirements cannot be analytically determined without some approximate representation of which index values will pass the screening tests. Hence, we make two simplifying assumptions:

1. On average, no more than s values of j from 1 to i and no more than s values of l from 1 to k survive screening to produce a non-negligible integral $(ij|kl)$. Both the triple-sort and canonical loop orderings require, as is assumed in this screening model, that j never be greater than i and l never be greater than k. For small molecules, $s \approx N$ and essentially every integral is computed, while for large molecules $1 \ll s \ll N$.

2. The s values of j that survive screening are generally located from $j = i - s$ to i. This assumption requires that basis functions be indexed in approximately the order of their centers when moving from one end of a molecule to the other.

For molecules containing many different types of atoms, the use of one screening parameter s is certainly a coarse simplification. Nevertheless, our experiments suggest that for many chemically bonded systems, the assumptions are reasonably accurate, and that s is typically of $\mathcal{O}(100)$ basis functions for molecules composed of first row atoms. The assumed location of the indices that survive screening is a coarse approximation but one that leads to convenient expressions of M and V that qualitatively represent the effect of screening.
As a simple application of this model, consider the number of integrals, N_{intg}, that must be computed:

$$N_{\mathrm{intg/screened}} = \sum_{i=1}^{N} \sum_{j=\mathtt{max}(1,i-s)}^{i} \sum_{k=1}^{i} \sum_{l=\mathtt{max}(1,k-s)}^{k} 1 \approx \frac{s^2 N^2}{2} \tag{17}$$

As is the case with all other expressions in this section, the approximate result is true only in the limit of $s \ll N$. The blocked algorithm sees a similar reduction:

$$M_{\mathrm{blocked/screened}} \approx \frac{6 s^2 N^2}{(I_c)^4} \tag{18}$$

$$V_{\mathrm{blocked/screened}} \approx \frac{6 s^2 N^2}{(I_c)^2} \tag{19}$$

The effect of screening on the row-blocked algorithm of Section 2 needs some elaboration. Each task requires six messages, resulting in

$$M_{\mathrm{row\_blocked/screened}} = 6 \sum_{i=1,I_c}^{N} \sum_{j=\mathtt{max}(1,i-s),I_c}^{i} \sum_{k=1,I_c}^{m} 1 \approx \frac{3 s N^2}{(I_c)^3} \tag{20}$$

where the value of m depends on triple-sort or canonical ordering but the final results do not. For V, the data requirements with screening are illustrated in Fig. 7; see Fig. 4 for the unscreened case. Each task must transfer $3(\min(k, s+1)) + 1$ D-matrix values in three messages. This requires that screening be performed on the j and l indices *before* data is transferred. In the triple-sort ordering, the triple permutation within each `compute(`i`,`j`,`k`,`l`)` operation necessitates screening tests on each permutation of the indices. The F-matrix rows are transferred at the end of the task, after screening has been performed. Hence, total communication volume is as follows; again, the results are independent of the choice of integral orderings.

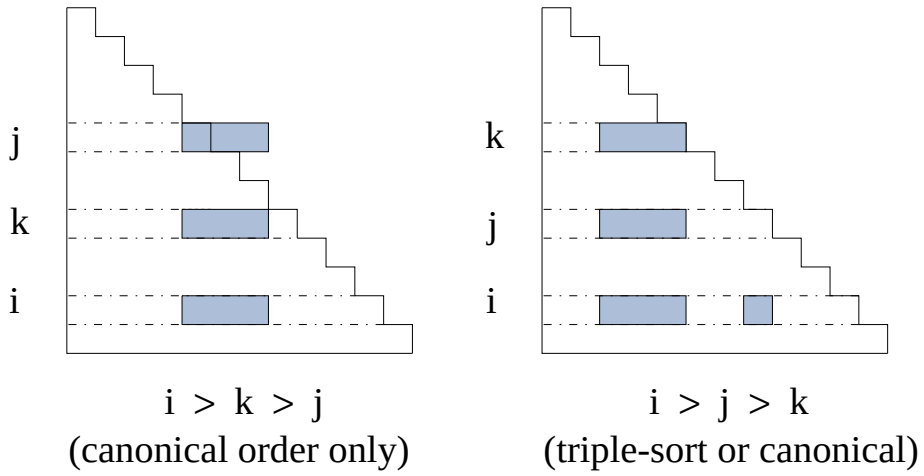

Figure 7: Data requirements for an (ijk) task, in the presence of screening with $s = 2$.

$$V_{\text{row\_blocked/screened}} = \sum_{i=1,I_c}^{N} \sum_{j=\texttt{max}(1,i-s),I_c}^{i} \sum_{k=1,I_c}^{m} (3\texttt{min}(k, s+1)+1) \approx 3s^2 \left(\frac{N}{I_c}\right)^2 \qquad (21)$$

In summary, we see that screening scales communication volumes in large molecules by $\mathcal{O}(s^2/N^2)$ and message counts by $\mathcal{O}(s/N)$. As computation costs are, in the limit, scaled by $\mathcal{O}(s^2/N^2)$, screening causes message counts to become a significantly greater contributor to total execution time. Screening also acts to make communications requirements of triple-sort and canonical orderings the same.

4.1 Dynamic Clustering

In Section 3, we showed that some components of $M_{\text{clustered}}$ (D_j, D_k, F_k) and $V_{\text{clustered}}$ (D_k, F_k) were identical to corresponding components of $M_{\text{row\_blocked}}$ and $V_{\text{row\_blocked}}$, while others needed to be corrected to allow for clustering. In addition, we determined that D message volumes were identical to F message volumes. These relationships also hold in the presence of screening. Hence, we need only determine the contribution of F_j and F_i to M and V and the contributions of D_i to M, all within the context of triple-sort ordering.

Consider first the contributions to M of F_j, F_i, and D_i. These contributions correspond to the latter three terms in Eqn. 14. Each term includes a summation over j: for F_j, the summation over j is explicit, while for F_i and D_j, the j summation is implicit in the non-P component of the `min` operation. Given our screening model, the lower limit of these summations must change from 1 to $\max(1, i-s)$, giving the expression:

$$M_{\text{clustered/screened}} = \frac{1}{2} M_{\text{row\_blocked/screened}} + M_{Fj/\text{screened}} + M_{Fi/\text{screened}} + M_{Di/\text{screened}} \qquad (22)$$

where

$$
\begin{aligned}
M_{Fj/\text{screened}} &= \sum_{i=1,I_c}^{N} \sum_{j=\texttt{max}(1,i-s),I_c}^{i} \texttt{min}(P, j/I_c) \\
M_{Fi/\text{screened}} &= \sum_{i=1,I_c}^{N} \texttt{min}(P, si/(I_c)^2) \\
M_{Di/\text{screened}} &= \sum_{i=1,I_c}^{N} \sum_{k=1,I_c}^{i} \texttt{min}(P, \texttt{min}[i-k, s]/I_c)
\end{aligned}
$$

Equation 15, the expression for message volume, can be generalized for screening into:

$$
V_{\text{clustered/screened}} = \frac{1}{3} V_{\text{row\_blocked/screened}} + V_{j/\text{screened}} + V_{i/\text{screened}} \tag{23}
$$

where

$$
\begin{aligned}
V_{j/\text{screened}} &= 2 \sum_{i=1,I_c}^{N} \sum_{j=\texttt{max}(1,i-s),I_c}^{i} \texttt{min} \left[\sum_{k=\texttt{max}(1,j-I_cP),I_c}^{j} (I_c)k, sj \right] \\
V_{i/\text{screened}} &= 2 \sum_{i=1,I_c}^{N} \texttt{min} \left[\sum_{k=\texttt{max}(1,i-klo),I_c}^{i} \texttt{min}(i-k, s)k, \sum_{j=\texttt{max}(1,i-s),I_c}^{i} js \right]
\end{aligned}
$$

In $V_{j/\text{screened}}$, the lower limit of the j summation has been modified to include the screening constraint. The `min` statement reflects two limits. If the number of processors is small, then each task performed on a processor for a given (i, j) pair will require segments of s columns that will overlap those required by other tasks on the same processor. In this "overlapping" limit, the message volume is unaffected by screening. This is the first argument in the `min` operation; it is identical to that in the unscreened case (Eqn. 15). On the other hand, if the number of processors is large, then the few tasks performed on each processor will have s column segments that do not overlap. In this limit, message volume is sI_c elements for each of the j/I_c values of k. This is the second argument in the `min` operation.

A similar analysis is applied in $V_{i/\text{screened}}$. The first argument in the `min` operation is the "overlapping" limit. This is identical to the no screening case (Eqn. 15), except that the lower limit in the k summation (klo) is changed. The klo in that limit has the unscreened value of $I_c(2P)^{\frac{1}{2}}$ if $P < (0.5)(s/I_c)^2$ or $I_c(P/s) - (0.5)(s/I_c)$ otherwise. The second argument is the large P limit, where we have sI_c elements for each (j, k) pair for a given i.

In the above expressions, the relationship between the clustered and the row-blocked algorithm as a function of P does not change substantially with screening. However, as with the row-blocked algorithm, screening makes the communications requirements of the clustered algorithm independent of triple-sort or canonical ordering.

There is some structure to the columns selected by screening in D and F. Since screening tests are similar for all basis functions in a shell of an atom and (to a lesser

extent) for all the basis functions of an atom, screening will tend to either process or ignore entire shell or atom submatrices in D and F (see Fig. 1). To process the `conditional_get`, the data transmitted is presumed column selected within the row-based data set. A more compact selection would be shell or atom selection.

4.2 Static Clustering

When evaluating the impact of screening on our static clustering algorithm, our first concern is to determine whether the number of integrals per supertask stays constant. In large molecules, it is generally the case that $s \ll p \leq N/2$. Hence, the number of integrals performed by that part of the supertask where $x = p$ is as follows, given our simplifying assumptions concerning screening.

$$\sum_{j=\mathtt{min}(2,p-s)\ \mathrm{by}\ 2}^{p} \sum_{k=1}^{p} min(1+s,k) \ + \ \sum_{i=p}^{N} \sum_{j=\mathtt{min}(1,i-2)\ \mathrm{by}\ 2}^{i} \mathtt{min}(1+s,p) \approx \frac{s^2 N}{2} + sp(N-p+1) \tag{24}$$

Adding the contribution for $x = N - p + 1$, we find that the number of integrals in a supertask is no longer independent of p. Hence, load imbalance becomes a problem, particularly when the number of processors is similar to the number of supertasks ($N/2$).

We have devised alternative static clustering algorithms that account for screening when constructing supertasks, but we have not been able to devise a single algorithm that applies in a variety of screening regimes. Hence, we conclude that our static clustering algorithm is probably not a robust method for allocating tasks to processors. In addition, we know that both the use of a single screening parameter s and the assumption that all integrals have the same cost are unrealistic approximations. We expect that tasks constructed using any algorithm based on these approximations will in practice differ considerably in cost. An accurate static clustering algorithm would appear to require a detailed analysis of the chemical nature of the basis functions, shells, or atoms involved in a particular system. In general, we doubt that the effort required to perform this analysis will be worthwhile.

5 Performance Analysis

The performance of a parallel algorithm is determined by both communication and computation requirements (as characterized in preceding sections in terms of M, V, and the number of integrals N_{intg}) and the characteristics of a particular MPP computer. Recall that the communication performance of a computer can be represented by parameters t_0 and t_1 (Eqn. 4) Computational performance can be related to the peak rate of floating point operations per second. However, in practice, such rates are hardly ever obtained. For the Fock matrix construction problem, the computation time $T_{compute}$ is best expressed as

$$T_{compute} = N_{\mathrm{intg}} t_{\mathrm{intg}} \tag{25}$$

where $t_{\rm intg}$ is the time per integral, a quantity that can be measured.

In the companion paper, timings of an actual code based on the ideas discussed here are presented for the Intel Touchstone Delta MPP computer. The values of t_0, t_1, and $t_{\rm intg}$ found to be appropriate for this computer are 300 microseconds (μsec), 1μsec, and 500μsec, respectively. These times depend on not only the inherent characteristics of the Delta but also the particular implementation of the communication operations and integral routines. However, they are representative of what can be obtained on today's production MPP computers.

In previous sections, M and V have been determined for blocked, row-blocked, and clustered algorithms and $N_{\rm intg}$ has been similarly defined for the screened and unscreened case. The performance of these algorithms can be conveniently represented by their *efficiency*, which is the ratio of the computation time to the sum of the computation time plus the total message time or:

$$Efficiency = \frac{N_{\rm intg} t_{\rm intg}}{N_{\rm intg} t_{\rm intg} + M t_0 + V t_1} \tag{26}$$

This quantity lies between 0 and 1 and when multiplied by the number of processors P gives the parallel speedup achieved by the code.

The efficiency of the various Fock matrix construction algorithms can now be determined as a function of N, s, and P. For example, with $N = 1000$ and $s = 100$, and assuming the communication parameters listed above, the efficiency predicted for the blocked, row-blocked, and clustered algorithms does not vary significantly for P up to 1000. However, it does vary strongly with I_c. With $I_c = 1$, the blocked algorithm achieves an efficiency of only 0.12, while for the row-blocked algorithm efficiency is 0.95, and the clucstered is as high as 0.98 for low P and 0.96 for high P. With $I_c = 10$, all algorithms achieve an efficiency of essentially 1.00. Note that these predicted efficiencies do not account for load imbalance and diagonalization.

These results of this analysis suggest that, as expected, the smaller number of messages generated by the row-blocked algorithm makes it significantly more efficient than the blocked algorithm when I_c is small. However, the stride I_c is clearly a more important determinant of performance. Uniformly increasing the stride on all three indices or, more generally, redefining i, j, and k to index collections of basis functions (i.e., shells, atoms, molecular fragments, etc.), seems to be the single most important strategy for improving efficiency. Our performance models predict that almost perfect parallel efficiencies should be achieved once the stride length is 10. A stride of ten is a typical value for the number of basis functions per atom in many basis sets of choice for atoms in the first row of the periodic table.

It is interesting to study the sensitivity of these results to machine parameters. Fig. 8 plots efficiency as a function of P when the computation/communication cost ratio is decreased by a factor of 100. This might correspond, as indicated in the Figure caption, to a 100-fold increase in computation speed ($t_{\rm intg} = 5\mu$sec, $t_0 = 300\mu$sec, $t_1 = 1\mu$sec), for example in a MPP machine of the future. Alternatively, it could represent a 100-fold decrease in communication speed ($t_{\rm intg} = 500\mu$sec, $t_0 = 30,000\mu$sec, $t_1 = 100\mu$sec), for example on a local area network (e.g., Ethernet). For interest, we include a replicated code, which avoids communication during computation but which must broadcast the

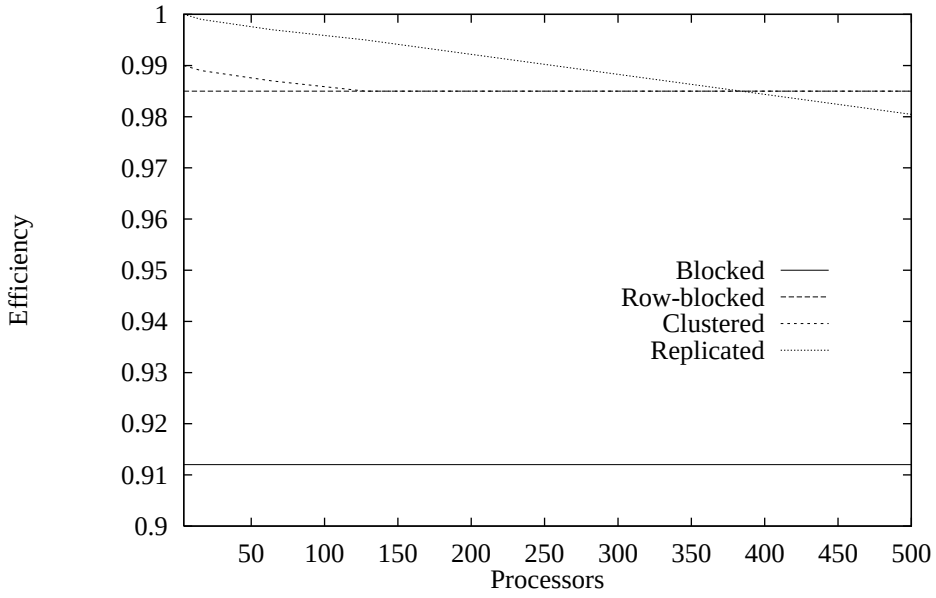

Figure 8: Predicted efficiency of four Fock matrix construction algorithms when $N = 1000$, $s = 100$, $I_c = 10$, and $t_{\text{intg}} = 5\mu$sec.

lower triangle of D and sum the partially constructed copies of the lower triangle of F located on each processor. We see that all methods continue to achieve good performance, although the row-blocked and clustered algorithms are now noticeably better than the blocked algorithm. We also see that the replicated code is not always the most efficient. At high values of P, it communicates large amounts of data that are never used because of screening. These results suggests that the algorithms described in this paper will be efficient even on loosely-coupled parallel computers, as long as a strides of 10 or larger are used.

Figure 8 might also correspond to a non-direct Hartree-Fock algorithm that exploits the considerable disk space of some MPP computers. For example, the 128 node IBM SP1 facility at Argonne National Laboratory has 1GB of dedicated disk space for each processor with an aggregate disk space of 128 GB. For problems with screening, the number of integrals grows as $(sN)^2/2$. If $s = 100$ and $N < 1750$, the integrals that survive screening could be computed in the first iteration and then saved onto local disk. In subsequent iterations, those integrals can be read off disk in such large batches that transfer time is bandwidth limited. With a realistic bandwidth of 1MB/sec, the cost of transfering an integral is about 8μsec. These are approximately the conditions of Figure 8.

At no time in this paper have we considered strides that were of different lengths for different `do-loops`. It is possible to generalize the formulas for M and V to this case. Suppose the possible combinations of different strides are restricted to those that have the same total local memory allocations for permanent and temporary storage of data

structures of D and F. Our analysis shows that under the circumstances considered in Fig. 8, it is the largest stride common to all the `do-loops` that determines efficiency.

6 Conclusion

We have presented several scalable and efficient parallel algorithms for the construction of the Fock matrix in SCF problems. The algorithms are scalable in that they allow the Fock and density matrices to be distributed over multiple processors. They are efficient in that they perform only a single pass through the integrals and allow for two levels of task grouping referred to as blocking and clustering which reduce both the total number and volume of interprocessor communications. These grouping techniques have been shown to be effective even when screening reduces computation costs from $\mathcal{O}(N^4)$ to $\mathcal{O}(N^2)$ integrals. We have characterized the effectiveness of the algorithms by developing mathematical models for communication costs as functions of problem size, number of processors, available memory, and a screening parameter.

Evaluations of these models with parameters typical of existing MPP computers show that all the algorithms proposed are highly efficient. The most efficient algorithms distribute the D and F matrices in blocks containing multiple rows. These algorithms require a communications library that in one message can transmit selected columns from these row-based data structures. The next most efficient algorithms use submatrices of D and F as their blocks. These algorithms can be implemented using a simple communications library able to transmit an entire submatrix with no need for column (or row or element) selection.

We have evaluated two alternative integral orderings: triple-sort and canonical. Our analysis suggests that triple-sort ordering is to be preferred over canonical. It requires one half as much storage and constructs a symmetrized Fock matrix. In the atypical case of applications with little screening, triple-sort performs less communication. Otherwise, communication costs are the same for both methods.

We have also examined two general classes of clustering technique: static and dynamic. Both schemes can adapt to varying amounts of memory, with communication requirements scaling inversely with available memory. We show that static clustering is superior from the point of view of communication requirements. However, it tends to suffer from load imbalances in the presence of screening. Hence, we recommend the more flexible dynamic clustering technique.

In a companion paper [1], a fully scalable code is presented that exploits the algorithms discussed here and, in addition, addresses other practical issues such as the distribution of all data structures of order N^2 (e.g., the overlap integrals used in screening), the development of a communications library for efficient data transfer, coding modifications to improve load balancing, and the diagonalization of the distributed Fock matrix. Empirical studies performed with this code confirm the general conclusions of this paper as does the work of Furlani and King [19].

Future work is being directed in three broad areas. First, the use of a electrostatic moment expansion for the calculation of the Coulomb interactions in the Fock matrix is a well-known way of reducing the number of individual integrals that must be calculated.

We are examining the implications of this approach for the algorithms discussed in this paper. Second, we are extending the current approach to compute higher order derivatives of the energy with respect to coordinates. Third, we are investigating alternatives to the use of conventional linear algebra methods for the diagonalization step. In particular, an energy minimization procedure in the parameter space of the basis function amplitudes appears promising on parallel computers.

Acknowledgments

We are grateful to John Garnett for his assistance in the early stages of this work, and to Ewing Lusk and Rick Stevens for insightful discussions. We thank Martyn F. Guest, David E. Bernholdt, Adrian T. Wong, Mark Stave, James Anchell, Anthony C. Hess, George L. Fann, Jaroslaw Nieplocha, Greg S. Thomas, and David Elwood of the Molecular Science Research Center, Pacific Northwest Laboratory, for providing access to code used in our experiments.

This work was performed under the auspices of the High Performance Computing and Communications Program of the Office of Scientific Computing, U.S. Department of Energy under contract W-31-109-Eng-38 with the University of Chicago which operates the Argonne National Laboratory and under contract DE-AC-6-76RLO 1830 with Battelle Memorial Institute which operates the Pacific Northwest Laboratory.

References

[1] Harrison, R. J., Guest, M. F., Kendall, R. A., Bernholdt, D. E., Wong, A. T., Stave, M., Anchell, J., Hess, A. C., Littlefield, R. L., Fann, G. L., Nieplocha, J., Thomas, G. S., Elwood, D., Tilson, J., Shepard, R. L., Wagner, A. F., Foster, I., Lusk, E., and Stevens, R. *submitted J. Comp. Chem.*, 1994.

[2] Roothaan, C., *Reviews of Modern Physics*, 23, 69, 1951.

[3] Almlöf, J., Faegri, K. Jr., Korsell, K., *J. Comp. Chem*, 3, 385, 1982.

[4] Harrison, R. J., Shepard, R, *For publication in Annual Review of Physical Chemistry*, 1994.

[5] Häser, M., Ahlrichs, R., *J. Comp. Chem*, 10, 104, 1989.

[6] Littlefield, R., and Maschhoff, K.,*Theor. Chim. Acta*, 84, 457, 1993.

[7] Shepard, R., *Theor. Chim. Acta*, 84, 343, 1993.

[8] Clementi, E., Corongiu, G., Detrich, J., Chin, S., Domingo, L., *Int. J. Quant. Chem.: Quantum Chem. Symp.*, 18, 601, 1984.

[9] Dupuis, M., Watts, J. D., *Theor. Chim. Acta*, 71, 91, 1987.

[10] Feyereisen, M., Kendall, R., Nichols, J., Dame, D., Golab, J., *J. Comp. Chem.*, 14, 818, 1993.

[11] Brode, S., Horn, H., Ehrig, M., Moldrup, D., Rice, J., Ahlrichs, R., *J. Comp. Chem.*, 14, 1142, 1993.

[12] Burkhardt, A., Wedig, U., Schnering, H. G. v., *Theor. Chim. Acta.*, 86, 497, 1993.

[13] Shirsat, R., Limaye, A., Gadre, S., *J. Comp. Chem.*, 14, 445, 1993.

[14] Lüthi, H., Mertz, M., Feyereisen, M., and Almlöf, J., *Intl J. Quant. Chem.*, 13, 160, 1992.

[15] Feyereisen, M., and Kendall, R., *Theo. Chim. Acta.*, 84, 289, 1993.

[16] Lüthi, H. P., Almlöf, *Theor. Chim. Acta.*, 84, 443, 1993.

[17] Guest, M., Sherwood, P., van Lenthe, J., *Theor. Chim. Acta.*, 84, 423, 1993.

[18] Colvin, M., Janssen, C., Whiteside, R., and Tong, C., *Theo. Chim. Acta.*, 84, 301, 1993.

[19] Furlani, T. R., King, H. F., *J. Comp. Chem.*, submitted for publication., 1994.

[20] Harrison, R., *Theo. Chim. Acta.*, 84, 363, 1993.

[21] A. Edelman, *J. Parallel and Distributed Computing*, 11, 328, 1991.

[22] S. L. Johnsson and C-T. Ho, *SIAM J. Matrix Anal. Appl.* 9, 419, 1988.