Draft May 16, 1994. For submission to the Journal of Computational Chemistry.

High Performance Computational Chemistry:(II) A Scalable SCF Program

Robert J. Harrison, Martyn F. Guest, Rick A. Kendall, David E. Bernholdt, Adrian T. Wong, Mark Stave, James L. Anchell, Anthony C. Hess, Rik J. Littlefield, George L. Fann, Jaroslaw Nieplocha, Greg S. Thomas, and David Elwood

Pacific Northwest Laboratory[1]*, P.O. Box 999, Richland WA 99352.*

Jeff Tilson, Ron L. Shepard, Albert F. Wagner, Ian T. Foster, Ewing Lusk and Rick Stevens

Argonne National Laboratory, Argonne, IL 60439.

We discuss issues in developing scalable parallel algorithms and focus in particular on the distribution, as opposed to the replication, of key data structures. Replication of large data structures limits the maximum calculation size by imposing a low ratio of processors to memory. Only applications which distribute both data and computation across processors are truly scalable.

The use of shared data structures that may be independently accessed by each process even in a distributed-memory environment greatly simplifies development and provides a significant performance enhancement. We describe tools we have developed to support this programming paradigm. These tools are used to develop a highly efficient and scalable algorithm to perform self-consistent field calculations on molecular systems. A simple and classical strip-mining algorithm suffices to achieve an efficient and scalable Fock-matrix construction in which all matrices are fully distributed. By stripmining over atoms we also exploit all available sparsity and pave the way to adopting more sophisticated methods for summation of the Coulomb and exchange interactions.

[1]Pacific Northwest Laboratory is operated for the U.S. Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RL0 1830.

1 Introduction

The objective of our research within our Federal High Performance Computing and Communications Initiative (HPCCI) Grand Challenge Applications project is to enable the use of very high performance computer architectures to solve Grand Challenge–class problems in computational chemistry. One goal in this effort is to develop *ab initio* electronic structure codes capable of modeling molecular systems of $O(10^{2-3})$ atoms and $O(10^{3-4})$ basis functions. To achieve this level of scalability will require use of massively parallel processors (MPP-s) and use of alternative numerical approaches. A recurring theme in our algorithm and software design and analysis is to develop techniques and tools that permit the development of scalable applications with explicit parallel constructs with only a minimal amount of extra effort.

In this article we discuss the design of scalable parallel algorithms that emphasize the cost-effective use of both processors and memory. We use the direct [1] closed-shell self-consistent field (SCF) method for molecules as an example. Efficient SCF computations are required for our research, and for more accurate theories. Also, SCF is representative of more sophisticated theories in using irregular data-access patterns and accumulation of results into large data structures. However, with current approaches, the computational cost of SCF grows at such a high rate with respect to problem size that even TERAFLOP computers[2] would permit only modest growth in problem size. To make more substantial improvements, it is necessary to incorporate more sophisticated algorithms with lower growth rates, for example, replacing an $O(N_{atom}^4)$ algorithm with one with an *effective* scaling of $O(N_{atom})$–$O(N_{atom}^2)$ [2, 3, 4, 5]. Parallelization of such algorithms will be the subject of

[2]Computers capable of a TERAFLOP, or 1012 floating point operations per second

future publications.

The first parallel self-consistent field (SCF) programs [6, 7, 8, 9, 10, 11, 12] were based on the "replicated data" model, in which each processor in the parallel system held its own complete copy of the Fock and density matrices. While this approach is simple and performs acceptably well for moderate-sized chemical systems, it does not scale well to either large chemical systems or massively parallel computers. Current parallel computers have only enough memory per node to replicate matrices of size 100–1000 square, and the high cost of memory relative to processors precludes adding memory. Instead, it is both necessary and cost-effective to distribute the Fock and density matrices across the memories of all processors.

Previous distributed SCF algorithms [13, 14] had the advantage of being based on regular communication patterns that were compatible with conventional "two-sided" message passing semantics (cooperative sender and receiver). They suffered, however, from inefficiencies arising from performing redundant computation, limited scalability and/or large process waiting times caused by load imbalance and frequent synchronization. In addition, development of these parallel programs based upon messages passing required significant effort beyond that of developing a sequential program with similar functionality. We wish to understand not merely how to write a fast and scalable SCF program, but to isolate the principles that make this task easier, and then apply those ideas to other algorithms of computational chemistry.

Below we continue our previous work [15] and explore an approach based on a novel communication strategy. The Fock matrix construction is parallelized using a dynamic load-balancing approach, as with the replicated data model [12]. However, the Fock and density matrices are distributed across processors and are accessed during the Fock-build using a "one-sided" remote-fetch/store/accumulate communication paradigm that is conceptually similar to shared memory. (We use the term "asynchronous global arrays" to describe this capability, which is discussed in more detail below.) This approach requires no extra computation (compared with the sequential algorithm) but is potentially communication-intensive.

We first describe briefly the key computational steps of the SCF non-linear optimization problem. Following this is discussion of a key concept in the design of efficient algorithms for current computer architectures (sequential or parallel) which uses the example of matrix multiplication. We apply the same techniques to the SCF problem and present our current algorithm. A simple performance model of the algorithm is discussed and results are presented.

2 Self-Consistent Field (SCF) — basic theory

A very brief summary of the non-relativistic SCF method in the Born-Oppenheimer approximation is given here, for more detailed references please refer to [16]. The restricted Hartree-Fock wavefunction for a closed-shell N-electron system is an antisymmetric product of N/2 doubly-occupied, orthonormal, molecular orbitals (MO-s). Each molecular orbital (ϕ_i) is expanded in a finite basis set (χ_μ) usually chosen as atomic centered functions (or atomic orbitals, AO-s) :

$$\phi_i(\underline{r}) = \sum_{\mu=1}^{N_{basis}} \chi_i(\underline{r}) C_{\mu i}, \tag{1}$$

where $C_{\mu i}$ are the coefficients that transform between the AO and MO basis sets, usually referred to as the MO coefficients. For simplicity we limit consideration to real MO coefficients.

The MO coefficients are chosen to minimize the energy subject to the orthonormality constraints. With the MO coefficients as parameters the SCF problem is a non-linear optimization problem with constraints. Alternatively [17], the parameters may be chosen to be orbital rotations, which results in an unconstrained non-linear optimization problem. The MO coefficients $C_{\mu i}$ are specified as a rotation of some initial orthonormal set of orbitals $C^{(0)}_{\mu i}$:

$$C = C^{(0)} e^{K}. \tag{2}$$

The matrix K is antisymmetric which ensures the rotation is unitary. The energy is invariant to rotation of either the occupied or unoccupied orbitals among themselves, so only the occupied-unoccupied block of K is determined by the minimization procedure.

We use letters $a, b, \ldots$ to denote unoccupied MO-s, $i, j, \ldots$ to denote occupied MO-s, $p, q, \ldots$ to denote arbitrary MO-s, and Greek letters to label AO-s. The energy and its derivative with respect to the orbital-rotation parameters k_{ai} may be expressed in terms of integrals over the molecular orbitals

$$E(K) = \sum_i (h_{ii} + F_{ii}), \tag{3}$$

and

$$\left.\frac{\partial E}{\partial k_{ai}}\right|_{K=0} = 4F_{ai}, \tag{4}$$

where h_{ij} are integrals over the one-electron operators in the Hamiltonian (e.g., kinetic energy and nuclear attraction operators) in the MO representation. Here F denotes the Fock matrix which elements are given by,

$$F_{pq} = h_{pq} + \frac{1}{2} \sum_k \left[2(pq|kk) - (pk|qk)\right]. \tag{5}$$

The two-electron integrals, $(pq|rs)$, arise from the electron-electron repulsion operator in the Hamiltonian and are defined as

$$(pq|rs) = \int d\tau_1 d\tau_2 \phi_p(\underline{r}_1)\phi_q(\underline{r}_1) r_{12}^{-1} \phi_r(\underline{r}_2)\phi_s(\underline{r}_2). \tag{6}$$

The MO-based equations are not very useful since the one- and two-electron integrals are computed in the AO basis. Defining the AO density matrix (D) as

$$D_{\mu\nu} = 2\sum_k C_{\mu k} C_{\nu k}, \tag{7}$$

the following equations are obtained for the energy

$$E(K) = \frac{1}{2}\sum_{\mu\nu} (h_{\mu\nu} + F_{\mu\nu})\, D_{\mu\nu}, \tag{8}$$

and the Fock matrix

$$F_{\mu\nu} = h_{\mu\nu} + \frac{1}{2}\sum_{\omega\lambda} [2(\mu\nu|\omega\lambda) - (\mu\omega|\nu\lambda)]\, D_{\omega\lambda}. \tag{9}$$

The two-electron integrals possess an eight-fold symmetry; interchange of the labels $\mu \leftrightarrow \nu$, or $\omega \leftrightarrow \lambda$, or $\mu\nu \leftrightarrow \omega\lambda$ leave the value of $(\mu\nu|\omega\lambda)$ unchanged. Use of these symmetries to avoid redundant computation results in each AO integral potentially contributing to six Fock matrix elements $(F_{\mu\nu}, F_{\mu\lambda}, F_{\mu\omega}, F_{\nu\lambda}, F_{\nu\omega}, F_{\omega\lambda})$.

An algorithm for the optimization of the SCF wavefunction is as follows.

1. Generate an initial set of orthonormal orbitals.

2. Generate the AO density from the orbitals.

3. Construct the Fock matrix.

4. From the Fock matrix or energy gradient determine improved orbitals.

5. Repeat from step 2 until converged.

The third step is the computational bottleneck on sequential computers. A conventional SCF program implements the fourth step by obtaining the new MO-s as the eigenvectors of the Fock matrix (possibly also employing damping, level-shifting and other methods [18]).

For N_{basis} basis functions there are $O(N_{basis}^4)$ two-electron integrals. Locality in the AO basis causes the number of integrals to decrease to approximately $O(N_{basis}^2)$ with the use of screening for very large, spatially-extended systems. Calculations with $O(10^2)$ basis functions are quite routine on scientific workstations and the largest calculations to date use $O(10^3)$ basis functions for systems with high symmetry on supercomputers.

The input/output (I/O) problem of storing and repetitively processing the two-electron integrals is avoided by computing the integrals as required [1]. Each integral may take several hundred or more floating point operations (FLOPs) to compute. This is particularly pertinent to current MPP architectures which generally provide very high computation rates and very poor I/O rates.

2.1 Replicated Data Parallel Algorithm

The replication of the Fock and density matrices within each processor of a distributed-memory parallel computer eliminates all communication during the two-electron Fock matrix contribution. Each processor computes part of the integral list adding them into its own local matrix. Subsequently, the complete Fock is obtained by combining the partial matrices with a global summation operation. The algorithm is perfectly parallel (assuming some rudimentary load balancing [8, 12, 19]) and the global summation of the Fock matrix can be done efficiently (e.g., on the Intel Touchstone Delta a global summation of 10^6 numbers takes about 3 seconds).

The poor scaling inherent in this approach is readily apparent. To hold two symmetric matrices of dimension N on each of P processors requires approximately $8PN^2$ bytes of memory. The size of calculation is constrained by the memory on each processor and the cost of the machine is dominated by memory rather than processors. By abandoning replicated data algorithms, we remove the rigid constraint on problem size and enable the usage of machines with a far more cost-effective ratio of processors to memory, a point raised by Hillis [20] in justifying the design of the CM-2.

Burkhardt et al [9] distributed the integral computation across many small Transputer-based nodes, keeping the Fock matrix on just one processor with a large memory. While this addresses the cost issue the algorithm is intrinsically not scalable, since accumulation of the integrals into the Fock matrix is a sequential bottleneck. Fully distributed algorithms [13, 14, 15] are required.

One key to understanding the design and performance of fully distributed algorithms is an understanding of how to manage efficiently the memory hierarchy of parallel computers. We consider such issues in the next section.

3 NUMA — Non Uniform Memory Access

NUMA refers to some regions of memory being more expensive to access than others. Consider for instance a standard RISC workstation. Its high performance stems from reliance on algorithms and compilers that optimize usage of the memory hierarchy formed by registers, on-chip cache, off-chip cache, main-memory and virtual memory. The programming of MIMD parallel computers (either shared or distributed memory) is united with the programming of sequential computers by the concept of NUMA. By focusing on NUMA, instead of the details of the programming model, parallel computation is seen to be different from sequential computation only in the essential difference of concurrent execution, rather than in nearly all aspects.

3.1 Stripmining or blocked algorithms

Consider the multiplication of two $N * N$ matrices

$$C_{ij} = C_{ij} + \sum_{k=1}^{N} A_{ik} B_{kj}. \tag{10}$$

A traditional vector supercomputer can fetch data from memory as fast as it can compute (which is why super computers are expensive – again it is the memory for which you are paying). Thus, it is not too ridiculous to use the simplest loop structure to implement the matrix multiplication (Figure 1).

The total number of memory references in the inner loop is $2N^3$, which is the same as the operation count. With the slower memory of low-cost workstations it is necessary to modify the algorithm so that most memory references are to blocks of the matrix kept in the fast cache. The blocking is achieved by partitioning or stripmining each loop (Figure 2) into N_{block} blocks.

The traffic between the cache and processor is still $2N^3$, but the traffic between the cache and memory is now just $2N_{block}N^2$, a reduction by a factor of N/N_{block}. If N_{block} is adjusted so that this ratio significantly exceeds the ratio of the bandwidths processor to cache, and cache to memory, then the matrix multiply can proceed essentially un-hindered by the slow memory (the cache must be big enough to hold the necessary block size). The issues are identical in parallel computation, except interprocessor bandwidth is compared with local-memory bandwidth, and the latencies associated with memory reference are more significant and must be incorporated into performance models.

The method we use to reduce communication in the parallel Fock matrix construction is identical to that used for this simple matrix multiplication example.

3.2 One-sided memory access

No emerging standards for parallel programming languages (notably just High Performance Fortran, HPF-1 [21]) provide extensive support for MIMD programming. The only truly portable MIMD programming model is message passing, for which a standard interface has been recently proposed [22] . It is, however, very hard to develop applications with fully distributed data structures using the message-passing model [13, 14]. What is needed is support for one-sided access to data structures (here limited to one- and two-dimensional arrays) in the spirit of shared memory. With some effort this can be done portably [23] and in return for this investment we gain a much easier programming environment that speeds code development and improves extensibility and maintainability.

We also gain a significant performance enhancement from increased asynchrony of execution of processes [24]. Message passing forces processes to cooperate (e.g., by responding to requests for a particular datum). Inevitably, this involves waiting for a collaborating process to reach the same point in the algorithm, which is only partially reduced by the use of complex buffering and asynchronous communication strategies. With a one-sided communication mechanism, where each process can access what it needs without explicit participation of another process, all processes can operate independently. This eliminates unnecessary synchronization and naturally leads to interleaving of computation and communication.

Most programs contain multiple algorithms some of which may naturally be task parallel (e.g., Fock matrix construction), and others that may be efficiently and compactly expressed as data parallel operations (e.g., evaluating the trace of a matrix product). Both types of parallelism must be efficiently supported.

4 Prototype support for distributed globally addressable arrays

Consideration of the requirements of the SCF algorithm discussed below, and also the parallel COLUMBUS configuration interaction program [25], second-order Møller-Plesset Perturbation theory and parallel Coupled-Cluster methods [26] led to the design and implementation of some preliminary tools [23] to support one-sided access to distributed one- and two-dimensional arrays. In this section we outline briefly the functionality provided by this library.

4.1 Programming model

The current GA programming model can be characterized as follows:

- MIMD parallelism is provided using a multi-process approach, in which all non-GA data, file descriptors, and so on are unique to each process.
- Processes can communicate with each other by creating and accessing GA distributed matrices, and also (if desired) by conventional message-passing.

- Matrices are physically distributed block-wise, either regularly or as the Cartesian product of irregular distributions on each axis.

- Each process can independently and asynchronously access any 2-D patch of a GA distributed matrix, without requiring cooperation by the application code in any other process.

- Several types of access are supported, include 'get', 'put', 'accumulate' (floating point sum-reduction), and 'get and increment' (integer). This list is expected to be extended as needed.

- Each process is assumed to have fast access to some portion of each distributed matrix, and slower access to the remainder. These speed differences define the data as being 'local' or 'remote', respectively. However, the numeric difference between 'local' and 'remote' access times is unspecified.

- Each process can determine which portion of each distributed matrix is stored 'locally'. Every element of a distributed matrix is guaranteed to be 'local' to exactly one process.

4.2 Supported operations

Each operation may be categorized as being either an implementation dependent primitive operation or constructed in an implementation independent fashion from primitive operations. Operations also differ in their implied synchronization. A final category is provided by interfaces to third party libraries.

The following are primitive, architecture dependent operations that are invoked synchronously by all processes:

- create an array, controlling alignment and distribution;
- destroy an array; and
- synchronize all processes.

The following are primitive operations that may be invoked in true MIMD style by any process with no implied synchronization with other processes:

- fetch, store and accumulate into rectangular patch of a two-dimensional array;
- gather and scatter array elements;
- atomic read and increment of an array element;
- inquiry about the location and distribution of the data; and
- access to local data to support and/or improve performance of application specific data-parallel operations.

The following are a set of BLAS-like data-parallel operations that have been developed on top of the primitive operations. Synchronization is included as a user convenience.

- vector operations (e.g., dot-product or scale) optimized to avoid communication by direct access to local data;
- matrix operations (e.g., symmetrize) optimized to reduce communication by direct access to local data; and
- matrix multiplication.

The following is functionality that is provided by third party libraries made available by using the GA primitives to perform necessary data rearrangement. The $O(N^2)$ cost of such rearrangement is observed to be negligible in comparison to that of $O(N^3)$ linear-algebra operations. These libraries may internally use any form of parallelism appropriate to the computer system, such as cooperative message passing or shared memory.

- standard and generalized real symmetric eigensolver; and
- linear equation solver (interface to SCALAPACK [27].

4.3 Sample code fragment

This interface has been designed in the light of emerging standards. In particular HPF [21] will certainly provide the basis for future standards definition of distributed arrays in FORTRAN. The basic functionality described above (create, fetch, store, accumulate, gather, scatter, data-parallel operations) all may be expressed as single statements using FORTRAN-90 array notation and the data-distribution directives of HPF. What HPF does not currently provide is random access to distributed arrays from within a MIMD parallel subroutine call-tree, and reduction into overlapping regions of shared arrays.

The following code fragment uses the FORTRAN interface to create an `n` by `m` double precision array, blocked in at least 10 by 5 chunks, which is zeroed and then has a patch filled from a local array. Undefined values are assumed to be computed elsewhere. The routine `ga_create()` returns in the variable `g_a` a handle to the global array with which subsequent references to the array may be made.

```
integer g_a, n, m, ilo, ihi, jlo, jhi, ldim

double precision local(1:ldim,*)
```

```
c
      call ga_create(MT_DBL, n, m, 'A', 10, 5, g_a)
      call ga_zero(g_a)
      call ga_put(g_a, ilo, ihi, jlo, jhi, local, ldim)
```

The above code is very similar in functionality to the following HPF-like statements

```
      integer n, m, ilo, ihi, jlo, jhi, ldim
      double precision a(n,m), local(1:ldim,*)
!hpf$ distribute a(block(10),block(5))
c
      a = 0.0
      a(ilo:ihi, jlo:jhi) = local(1:ihi-ilo+1, 1:jhi-jlo+1)
```

The difference is that this single HPF assignment would be executed in a data-parallel fashion, whereas the global array put operation will execute in MIMD parallel mode such that each process may reference different array patches.

5 The distributed SCF algorithm

Our initial distributed SCF prototype code [15] parallelized only the two-electron contribution to the Fock matrix construction. This effort was successful in that an acceptably efficient parallel Fock matrix construction was achieved. It was apparent, however, that the remainder of the program, which was still using replicated data methods, needed complete rewriting. There were several problems: the distributed data tools we had used did not accommodate efficient data-parallel operations; the distribution of tasks was overly complex, causing load balancing problems; excess data was being moved in sparse problems; and the overall parallel scalability was not as good as desired.

The design of the current algorithm follows the analysis used for the matrix-multiply algorithm above. The cost of accessing an element of the global density or Fock matrices must be offset by using this element multiple times. To achieve this re-use of data we simply stripmine the four-fold loop over basis functions. Since the computation is a quartic function of the block size while the communication is only a quadratic function, small block sizes suffice to make the computation time dominate the communication time. This is the same principle independently used by Furlani and King [14], but, in constrast to the complexities of their message-passing algorithm, the support for "one-sided" access to distributed arrays enables a very simple, efficient and readily modified implementation.

We presently stripmine by grouping basis functions according to their (usually atomic) centers. While this has the disadvantage that the granularity is fixed, the granularity is sufficient for our initial target machines (Intel Touchstone Delta, Kendall Square Research and IBM SP1), and the scheme has the advantage that the sparsity may be used in the outer stripmining loops. The simplest parallel loop structure is presented in Figure 3.

The four loops over `iat`, `jat`, `kat`, and `lat` determine unique quartets of atoms, and the outermost two `IF` blocks take advantage of sparsity using the Schwarz inequality (the screening information is compressed and may be distributed). Parallelism with full dynamic load-balancing is introduced by comparing a sequential count of interacting atomic quartets (`ijkl`) against the value of a shared counter accessed by calling the function `next_task()`. Each task requires fetching up to six blocks of the density matrix and accumulating into the corresponding blocks of the Fock matrix.

Following our previous analysis [15], communication may be reduced by up to a factor of two by defining a task as all `lat` for given `iat, jat,` and `kat`. With this task definition the `ij, ik,` and `jk` blocks of the matrices are common to all atomic quartets in a task. Caching of the blocks is preferred to pre-fetching for both its simplicity, and so that communication takes full advantage of sparsity. The larger tasks also eliminate a potential bottleneck accessing the shared counter. However, the available parallelism is limited by this choice and the largest task size increases linearly with the number of atoms in the molecule which causes load-balancing problems. Thus, we actually define a task to be for given `iat, jat,` and `kat` up to five values of `lat`, five being a comprise between a large value for optimal caching and a small value for fine grain load balance. Finally, contention for access to global array elements is reduced and load-balancing is further improved by reversing the loop order and randomizing the order of atoms in the input. The final algorithm is given in Figure 4.

Since the global-array tools provide direct support for both data-parallel and task-parallel operations it was straightforward to parallelize the remainder of the SCF program. Thus, all computational steps of complexity greater than $O(N_{atom})$ are parallelized (essentially only the input and output phases are sequential). Efficient parallel matrix multiply operations are implemented directly using the global-array tools, while other operations (e.g., matrix diagonalization) internally convert to the data-layout required by the parallel linear algebra routines. In contrast to the Fock matrix construction, the diagonalizer is parallelized by using conventional message-passing instead of the global array routines. This strategy, which is feasible because of the regularity of the linear algebra algorithms, produces somewhat higher efficiency.

We emphasize that the combination of "asynchronous global array" access and conventional message passing produces a hybrid communication strategy that allows optimizing different parts of the application in different ways. Those portions of the application that are not communication intensive but have many different task sizes and unpredictable communication patterns are best implemented with asynchronous arrays; conventional message passing may be more appropriate to those portions that are communication-intensive but regular. We anticipate that such hybrid schemes will become increasingly common.

5.1 Performance Model

Let the basis functions on each atom overlap on average with functions on α other atoms. The factor α (between 1 and N_{atom}) is determined by the molecular geometry, diffuseness of the basis set, and the integral screening thresholds. There will be $\frac{(\alpha N_{atom})^2}{8}$ interacting atomic quartets. If each integral takes β seconds to compute, and perfect load balance (see below) is achieved then the total computational cost on P processors is approximately

$$T_{compute} = \frac{(\alpha N_{atom})^2}{8} \frac{\beta}{P} \left(\frac{N_{basis}}{N_{atom}} \right)^4 . \tag{11}$$

On average, about four blocks of the density and Fock matrices must be accessed for each quartet. The communication cost is thus approximately (neglecting possible contention and queuing)

$$T_{communicate} = \frac{(\alpha N_{atom})^2}{8} \frac{8}{P} \left(t_0 + t_1 \left(\frac{N_{basis}}{N_{atom}} \right)^2 \right) , \tag{12}$$

where t_0 and t_1 are the latency in seconds and transmission cost in seconds per floating-point number. The ratio of computation to communication is readily computed as

$$\frac{T_{compute}}{T_{communicate}} = \frac{\beta \left(\frac{N_{basis}}{N_{atom}} \right)^4}{8 \left(t_0 + t_1 \left(\frac{N_{basis}}{N_{atom}} \right)^2 \right)}, \tag{13}$$

Appropriate values for the Intel Touchstone Delta and a double-zeta plus polarization basis set

- $\beta = 0.0001$ seconds/integral
- $\frac{N_{basis}}{N_{atom}} = 10$
- $t_0 = 0.0003$ seconds
- $t_1 = 10^{-6}$ seconds/word

Substituting these numbers we obtain

$$\frac{T_{compute}}{T_{communicate}} = \frac{1}{8(0.0003 + 0.0001)} = 312.5 \tag{14}$$

This high ratio of computation to communication predicts very high parallel efficiency independent of sparsity up to $O(N_{atom}^2)$ processors. With the current granularity the communication cost is primarily due to latency. The large ratio between computation and communication justifies the assumption that communication contention is not significant. Contention will only become an issue if the number of processors approaches (on mesh architectures such as the Delta) the square of this ratio. Queuing for data access is empirically observed to be not significant for the algorithm of Figure 4. We have also investigated various strategies for randomization of tasks which rigorously eliminated queuing, however we obtained slightly degraded performance due to fluctuations in load balance. Excellent load balance is ensured by arranging for large tasks to occur first, limiting the maximum task size and by using full dynamic load balancing.

5.2 Performance

Several test calculations have been executed to demonstrate and explore the performance of the new program. The molecules, which were near their equilibrium geometries, and basis sets are briefly described in Table 1. Full details of basis sets and geometries are available from the authors upon request.

Figure 5 displays the speedups obtained for the two-electron Fock-matrix construction for these systems on the Intel Touchstone Delta. The single processor times for all but the two smallest systems were estimated by assuming that the calculations on the smallest number of processors used were executing at 99% efficiency. We observe that the series of similar molecules C_2H_6, C_4H_{10}, C_8H_{18} demonstrate best speedups of 31, 81, and 367, respectively, which are very close to $N_{atom}^2/2$. This finite speedup is due to the available parallelism being exhausted. Speedup degrades before this number of processors are used because as the number of tasks per processor diminishes the load balance worsens. For these small systems better asymptotic performance could be obtained by varying the maximum task size with the number of processors, so that in the limit of many processors each processor evaluates the interactions of just one quartet of atoms.

Systems with more interactions (e.g., the 28 atom cluster $Si_8O_{12}H_8$, as opposed to the chain C_8H_{18}) or more atoms demonstrate excellent speedup. The largest system, $Si_{16}O_{25}H_{14}$ in a 6-31g* basis, obtains a speedup of 496 on 512 processors, an efficiency of 97%. It is not yet apparent why the efficiency is not higher still, as would be expected from the simple performance model above. One possible cause is that the model is too simplistic in its use of a fixed average value of 10 for the number of basis functions per atom.

Next to the Fock-matrix construction, the other major step is the diagonalization, which while running in parallel, obtains an effective speedup of only 6. It is this that results in the speedup of the entire SCF calculation being only 438 on 512 processors, an efficiency of 86%. The parallel efficiency of the diagonalization increases as the matrix size increases, however the $O(N^3)$ scaling of the diagonalization is similar to the scaling of integral evaluation in

```
DO i = 1, N
  DO j = 1, N
    sum = Cij
    DO k = 1, N
      sum = sum + Aik * Bkj
    ENDDO k
    Cij = sum
  ENDDO j
ENDDO i
```

Figure 1: A simple matrix-multiplication algorithm.

Molecule	Basis	Atoms	Functions
C_2H_6	C(5s2p1d)/H(2s1p)†	8	64
C_4H_{10}	C(5s2p1d)/H(2s1p)†	14	118
C_8H_{18}	C(5s2p1d)/H(2s1p)†	26	226
$Si_8O_{12}H_8$	6-31g‡	28	228
$Si_8O_{12}H_8$	6-31g*‡	28	348
$Si_{16}O_{25}H_{14}$	6-31g‡	55	461
$Si_{16}O_{25}H_{14}$	6-31g*‡	55	707

† ‡

Table 1: Systems used to study performance of the SCF program.

```
DO iblock = 1, Nblock
  DO jblock = 1, Nblock
    Load block of C into cache
    DO kblock = 1, Nblock
      Load A and B blocks into cache

      DO i in iblock
        DO j in jblock
          sum = Cij
          DO k in kblock
            sum = sum + Aik * Bkj
          ENDDO k
          Cij = sum
        ENDDO j
      ENDDO i
    ENDDO kblock
  ENDDO jblock
ENDDO iblock
```

Figure 2: A stripmined matrix-multiplication algorithm.

```
task = next_task()
ijkl = 0
DO iat = 1, Natom
  DO jat = 1, iat
    IF (T(iat,jat) .gt. tol1) then
      DO kat = 1, iat
        lat_hi = kat
        IF (kat .eq. iat) lat_hi = jat
        DO lat = 1, lat_hi
          IF (T(iat,jat)*T(kat,lat) .gt. tol2) then
            IF (ijkl .eq task) then

              Fetch atomic blocks of density matrix
              Dij, Dik, Dil, Djk, Djl, Dkl

              Compute integrals and form Fock matrix

              Update atomic blocks of Fock matrix
              Fij, Fik, Fil, Fjk, Fjl, Fkl

              task = next_task()

            ENDIF
            ijkl = ijkl + 1
          ENDIF
        ENDDO lat
      ENDDO kat
    ENDIF
  ENDDO jat
ENDDO iat
```

Figure 3: Pseudo-code representing the loop structure of the simplest stripmined parallel two-electron Fock matrix construction.

```
task = next_task()
ijkl = 0
DO kat = Natom, 1, -1
  DO iat = Natom, kat, -1
    DO jat = iat, 1, -1
      IF (T(iat,jat) .gt. tol1) then
        lat_hi = kat
        IF (kat .eq. iat) lat_hi = jat
        DO lat_lo = 1, lat_hi, 5
          IF (ijkl .eq task) then
            DO lat = lat_lo, MIN(lat_lo+4, lat_hi)
              IF (T(iat,jat)*T(kat,lat) .gt. tol2) then

                Fetch only if changed Dij, Dik, Djk
                (also storing corresponding Fock elements)

                Fetch Dil, Djl, Dkl

                Compute integrals and form Fock matrix

                Update Fil, Fjl, Fkl

              ENDIF
            ENDDO lat
            task = next_task()
          ENDIF
          ijk = ijk + 1
        ENDDO llo
      ENDIF
    ENDDO jat
  ENDDO iat
ENDDO kat
```

Figure 4: Pseudo-code representing the loop structure of the final stripmined parallel two-electron Fock matrix construction.

Figure 5: Speedup of two-electron Fock matrix construction for the test systems versus number of processors used on the Intel Touchstone Delta.

large molecules. Once improved algorithms are adopted for the Fock matrix construction [2, 3, 4, 5] the diagonalization will be dominant. Several approaches have been proposed for eliminating this bottleneck [17, 28]. We are adopting a variation of the second-order convergent approach proposed by Shepard [17], in part because of the wide range of properties that may be computed from the orbital Hessian.

6 Conclusions

A simple and classical strip-mining algorithm suffices to achieve an efficient and scalable Fock-matrix construction in which all matrices are fully distributed. Since the computation is a quartic function of the block size while the communication is only a quadratic function, small block sizes suffice to make the computation time dominate the communication time. A simple performance model that takes into account the cost of integral evaluation, the volume of communication, and the latency and bandwidth of communication predicts that a constant efficiency of about 99% (for the Fock matrix construction) is achieved for the number of processors less than approximately the square of the number of atoms. An efficiency of 97% is measured for a large molecular system on 512 processors. A production version of this algorithm would include dynamic adjustment of the granularity in response to the basis set size and machine performance parameters. Far greater gains are realizable, however, by first pursuing alternative algorithms [2, 3, 4, 5].

The current programming model, mixing globally-addressable arrays with message passing is both portable and efficient. In particular, machines such as the Kendall Square Research and the Cray T3D which have hardware support for shared memory should prove particularly effective in this model.

7 Acknowledgments

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 DE-AC066-76RLO 1830 with Battelle Memorial Institute which operates the Pacific Northwest Laboratory, and contract W-31-109-ENG-38 with the University of Chicago which operates the Argonne National Laboratory. This work has made extensive use of software developed by the Molecular Science Software Group of the Molecular Science Research Center, PNL, which is part of the Environmental and Molecular Science Laboratory. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory.

References

[1] Almlöf J, Faegri K, and Korsell K. 1982. *J. Comp. Chem.*, 3:385.

[2] Panas I and Almlöf J. 1992. *Int. J. Quant. Chem.*, 42:1073–1089.

[3] Dovesi R, Saunders VR, and Roetti C. *Crystal92 user's manual.* University of Torino, Torino, 1992.

[4] Ringald MN, Belhadj M, and Friesner RA. 1990. *J. Chem. Phys.*, 93:3397–3347.

[5] Vahtras O, Almlöf J, and Feyereisen MW. 1993. *Chem. Phys. Lett.*, 231:514.

[6] Clementi E, Corongiu G, Detrich J, Chin S, and Domingo L. 1984. *Int. J. Quant. Chem. Sym.*, 18:601–618.

[7] Dupuis M and Watts JD. 1987. *Theor. Chim. Acta*, 71:91–103.

[8] Guest MF, Smith W, and Harrison RJ. 1992. *Chem. Design Auto. News*, 7:12–18.

[9] Burkhardt A, Wedig U, and v. Schnering HG. 1993. *Theor. Chim. Acta*, 86:497–510.

[10] Lüthi HP, Mertz JE, Feyereisen MW, and Almlöf JE. 1992. *J. Comp. Chem.*, 13:160–164.

[11] Otto P and Früchtl H. 1993. *Computers in Chemistry*, 17:229–239.

[12] Feyereisen M and Kendall RA. 1993. *Theor. Chim. Acta*, 84:289–299.

[13] Colvin ME, Janssen CL, Whiteside RA, and Tong CH. 1993. *Theor. Chim. Acta*, 84:301–314.

[14] Furlani TR and King HF. 1993. *J. Comp. Chem.*, submitted for publication.

[15] Foster IT, Tilson JL, Wagner AF, Shepard R, Harrison RJ, Kendall RA, Littlefield RJ, Bernholdt DE, and Anchel J. 1994. *J. Comp. Chem.*, ??:??

[16] Szabo A and Ostlund NS. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory. 1st Ed. Revised.* McGraw-Hill, Inc., New York, 1989.

[17] Shepard R. 1993. *Theor. Chim. Acta*, 84:343–351.

[18] P. Pulay. 1982. *J. Comp. Chem.*, 3:556.

[19] Harrison RJ and Kendall RA. 1991. *Theor. Chim. Acta*, 79:337–347.

[20] Hillis WD. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[21] High Performance Fortran Forum. Technical Report Version 1.0, Rice University, 1993.

[22] The MPI Forum. In *Proceddings of Supercomputing '93*, pages 878–883. IEEE computer Society Press, Los Alamitos, California, 1993 November, 1993.

[23] Nieplocha J, Harrison RJ, and Littlefield RJ. Asynchronous global arrays, Supercomputing 1994, submitted for publication.

[24] Carriero N and Gelernter D. *How To Write Parallel Programs. A First Course.* The MIT Press, Cambridge, MA, 1990.

[25] Schuler M, Kovar T, Lischka H, Shepard R, and Harrison RJ. 1993. *Theor. Chim. Acta*, 84:489–509.

[26] Rendell AP, Guest MF, and Kendall RA. 1993. *J. Comp. Chem.*, 14:1429–1439.

[27] SCALAPACK, scalable linear algebra package, code and documents available through `netlib`.

[28] Pollard WT and Friesner RA. 1993. *J. Chem. Phys.*, 99:6742.