Theor Chim Acta (1993) 84: 301–314

Theoretica Chimica Acta

Parallel direct SCF for large-scale calculations

M. E. Colvin, C. L. Janssen, R. A. Whiteside*, and C. H. Tong
Center for Computational Engineering, Sandia National Laboratory, Livermore, CA 94551, USA

Received October 1, 1991/Accepted June 29, 1992

Summary. Quantum chemistry has become an essential tool in many areas of chemical research; however, quantum chemistry is not yet playing a role in many exciting new chemical disciplines, such as medicinal chemistry and materials science, where the size of the chemical systems has been too large to study using *ab initio* chemical methods. The development of massively parallel supercomputers offers the potential to predict properties relevant to a variety of problems in these burgeoning new fields. The goal of this project is to develop a set of parallelized "production codes" for initially a relatively limited set of methods. As a key part of this project we are experimenting with the use of modern programming languages and methodologies to make these programs both portable and reusable. This paper describes the development of a massively parallel direct SCF program, MPSCF. For systems over a few hundred basic functions, MPSCF running on 256 nCUBE processors performs nearly as well as Gaussian 90 running on a single processor Cray Y-MP. On the next generation of parallel computers, such as the Intel Touchstone Delta, MPSCF should allow the SCF calculations on chemical systems too large for vector supercomputers.

Key words: Parallel direct SCF – Quantum chemistry – MPSCF – LAN

1 Introduction

Quantum chemistry has become an essential tool in many areas of chemical research; however, quantum chemistry is not yet playing a role in many exciting new chemical disciplines, such as medicinal chemistry and materials science, where the size of the chemical systems has been too large to study using *ab initio* chemical methods. Yet the relatively low accuracy needed to provide useful results and their tremendous practical importance make these very promising new applications for quantum chemistry. The development of massively parallel supercomputers offers the potential to predict properties relevant to a variety of problems in these burgeoning new fields. We envision two broad classes of

* *Present address*: Hypercube Inc. c/o Autodesk Inc, Sausalito, CA, USA

parallel computer becoming important in the next few years. The first are the massively parallel supercomputers such as the nCUBE 2 and the Intel Touchstone which can outperform vector supercomputers for some applications. The other will be implicit in the large networks of high-performance workstations located in many research institutions. Harnessed together to work on a single calculation, these local area networks (LANs) constitute a very large, as yet untapped resource. However, the relatively limited bandwidth of current LANs will limit their performance in the near future. Both of these classes of parallel computers involve unique programming challenges and since our goal is to perform calculations on extremely large chemical systems we have focused our development effort on the massively parallel supercomputers.

The potential of parallel computers has not gone unnoticed by quantum chemists. Since the mid-1970's there have been several efforts to parallelize quantum chemistry programs [1, 2]. However, due to the experimental nature of the early parallel computers, these programs were mainly "proofs of principle" and not alternatives to production programs on serial computers. With advances in parallel processing hardware, there has been a growing interest among quantum chemists. A recent conference on parallel quantum chemistry [3] attracted more than thirty participants who presented papers on parallel SCF and post-SCF methods.

The goal of this project is to develop a set of parallelized "production codes" for initially a relatively limited set of methods. As a key part of this project we are experimenting with the use of modern programming languages and methodologies to make these programs both portable and reusable. This paper describes the first step in this effort, the massively parallel direct SCF program, MPSCF.

2 Algorithms

Developing algorithms for parallel computers is particularly challenging due to the fact – known as "Amdahl's law" – that the parallel speedup is at best the inverse of the fraction of the code that is serial. This fact frequently undermines the effectiveness of automatic parallelizing compilers, especially for large heterogeneous programs typical of quantum chemistry applications. Hence, parallel programming is not automatic and involves manually breaking up algorithms into individual modules to run on different processors. Although there are a large number of different parallel architectures available, at this time message-passing, multiple-data multiple-instruction (MIMD) [4] architectures are the most flexible and extensible. For this reason we are targeting this general class of machine with the specific goal of using the nCUBE/2 and the Intel i860 and Intel Delta computers. (See Appendix for a detailed description of these computers.)

The first quantum chemical application that we have parallelized is the self-consistent field (SCF) method. A simplified version of the computational steps involved in a direct SCF are shown in Fig. 1. The SCF method is an iterative procedure involving the repeated formation and diagonalization of a matrix of order up to a few thousand for our chemical applications. The iterative procedure naturally separates into two portions, the matrix operations (steps 2, 4, & 5) and the Fock matrix formation which will be separately discussed in the following paragraphs. As implemented, MPSCF is slightly more sophisticated than the algorithm described in Fig. 1. In particular, rather than calculate the Fock matrix directly as shown in step 3, only the change in the Fock matrix

Steps in SCF calculation:	Computational complexity in terms of # basis functions, N
1) Generate initial guess at SCF vector $\boldsymbol{C}$.	
2) Form density matrix $\boldsymbol{P} = \boldsymbol{C}^{\dagger}\boldsymbol{C}$.	$O(N^3)$
3) Calculate two electron integrals and combine with $\boldsymbol{P}$ to form Fock matrix $\boldsymbol{F}$: $$\boldsymbol{F}_{\mu\nu} = \sum_{\lambda\sigma} \boldsymbol{P}_{\lambda\sigma}((\mu\nu \mid \lambda\sigma) - \tfrac{1}{2}(\mu\lambda \mid \nu\sigma))$$	$O(N^4)$
4) Transform $\boldsymbol{F}$ using the inverse root of the overlap matrix: $$\boldsymbol{F}^{\dagger} = \boldsymbol{S}^{-1/2}\boldsymbol{F}\boldsymbol{S}^{-1/2}.$$	$O(N^3)$
5) Calculate eigenvectors of $\boldsymbol{F}^{\dagger}$ and backtransform to form new $\boldsymbol{C}$.	$O(N^3)$
6) Stop if $\boldsymbol{C}$ is converged, else go to 2.	

Fig. 1. Steps in the two-electron portion of a self-consistent field (SCF) calculation

is calculated:

$$F_{\mu\nu}^{\text{new}} = F_{\mu\nu}^{\text{old}} + \sum_{\lambda\sigma} \Delta P_{\lambda\sigma}[(\mu\nu \mid \lambda\sigma) - \tfrac{1}{2}(\mu\lambda \mid \nu\sigma)]$$

The advantage of this approach is that as the calculation converges the elements of ΔP becomes small. This, along with methods for estimating the upper bounds of two-electron integrals (discussed later) drastically reduces the computational effort of forming the Fock matrix [5].

Additionally, MPSCF includes the DIIS method developed by Pulay for accelerating SCF convergence [6]. The DIIS procedure involves extrapolating the Fock matrix as a linear combination of Fock matrices that minimize the error matrix:

$$\varepsilon = \boldsymbol{EPS} - \boldsymbol{SPF}$$

Computationally, DIIS involves several matrix multiplies and the solution of a set of linear equations of very low order (typically 5).

A natural way to parallelize SCF is to distribute the matrix formation and manipulation. An important initial consideration is whether to distribute the storage of the matrices or just the computational load. For a direct SCF calculation at minimum both the Fock and density matrices must be available on the processing nodes. This requires nbasis*nbasis (nbasis = #basis functions) double precision words to be stored. Since we are interested in running calculations with up to several thousand basis functions we will not be able to fit complete copies of these matrices on each node of even the largest parallel computers now available. There are many possible schemes for evenly distributing the storage of square matrices. For example, in a column distribution scheme each processor would be assigned one or more columns of the matrix. In block distribution, each processor is given a set of submatrices obtained by dividing the matrix into sub-blocks. However, the additional need to both minimize and evenly balance the computational load adds more constraints. The expressions for the two-electron integrals over basis functions in the same shell block involve many common sub-expressions. The efficiency of calculating these terms can be greatly improved by calculating such groups of integrals as a batch. As shown in the equation in Fig. 1, each Fock matrix element, $F_{\mu\nu}$, contains contributions from the integrals $(\mu\nu \mid \lambda\sigma)$ and $(\mu\lambda \mid \nu\sigma)$. Therefore, if shell blocks of the Fock matrix are assigned to the processors so that all μ's and all ν's in a particular shell block are on the

same processor the integral efficiency is greatly increased. Moreover, such a distribution of the density matrix elements facilitates the estimation of upper bounds on the Fock matrix elements (*vide infra*). For many of the matrix operations, however, the most efficient algorithms currently available require the matrices distributed by column. Hence, in the present version of MPSCF, the block-distributed matrices are converted to column distribution for some of the matrix manipulations.

Since the size and computational effort associated with each shell block is different, the load balancing of the Fock matrix formation becomes an issue. For this type of problem optimal load balancing is very difficult, formally equivalent to the knapsack problem known to be NP complete [7]. However, the number of shell blocks is roughly proportional to the number of basis functions squared, while the maximum number of processors we can efficiently use it less than half of the number of basis functions (see section on parallel matrix diagonalization). Hence, even the simple method placing equal numbers of shell blocks (albeit all of different size) on each processor yields adequate load balancing for the benchmark reported herein which included only *s* and *p* type shell blocks. For cases involving larger angular momentum or derivative integral evaluation, the variance in shell block size will be much larger so that a more careful analysis is needed. Work is underway to examine the effect that a more sophisticated distribution scheme can have on load balancing for high angular momentum cases.

Since each Fock matrix element contains contributions from every density matrix element, some form of communication is required between every pair of processing elements during each SCF iteration. Many "complete shuffle" algorithms are possible, but the simplest and least hardware dependent is a systolic loop. All of the processors are assumed to be connected in a one-dimensional loop (which can easily be embedded in a hypercube or mesh topology) and data packets are passed around the ring in as many systolic steps as there are processors.

For the Fock matrix formation each processor sends out its density matrix blocks and initially empty message buffers to accumulate contributions to its locally held set of Fock matrix blocks (see Fig. 2). To illustrate this procedure, consider the set of blocks $\Delta P_{\lambda\sigma}$ and $F_{\lambda\sigma}$ arriving on the processing node assigned the $\mu\nu$ blocks. If the integral estimation procedure indicates that they are significant, the processor computes the coulomb integrals $(\mu\nu \mid \lambda\sigma)$ and the exchange integrals $(\mu\lambda \mid \nu\sigma)$ and $(\mu\sigma \mid \nu\lambda)$. Note that if there are coincidences in

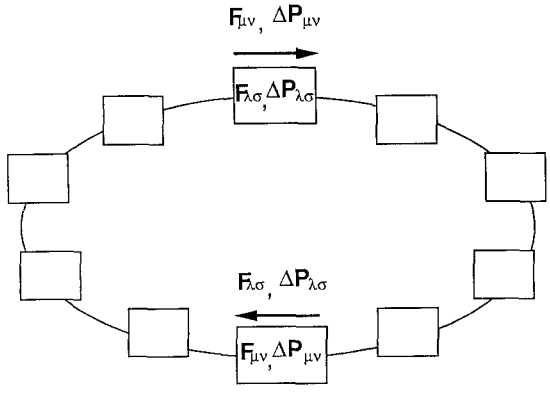

Fig. 2. Formation of the Fock matrix on a systolic loop of processors. Each node makes contributions from its locally held density matrix elements and integrals to the Fock matrix elements being passed around the loop. Additionally, the node makes contributions from the density matrix elements being passed around the loop to its locally stored Fock matrix elements

the shell indices, some of these blocks are redundant. From these integral blocks, contributions must be made to the locally held Fock matrix elements:

$$F_{\mu\nu} \leftarrow P_{\lambda\sigma}(\lambda\sigma \mid \mu\nu)$$

and the Fock elements being communicated around the ring:

$$F_{\lambda\sigma} \leftarrow P_{\mu\nu}(\mu\nu \mid \lambda\sigma)$$

After all of the ΔP and F blocks have been sent around the processor ring, the full two-electron Fock matrix has been formed. At this point the one-electron terms are added and the Fock matrix is ready for transformation and diagonalization.

The computer time required for the formation of the Fock matrix is dominated by the calculation of the two electron integrals. The actual number of operations to evaluate an integral is very dependent on the basis function angular momentum and the exact algorithm chosen. For the purpose of this simple analysis we will assume that an average of σ double precision (DP) floating point operations is required. Hence, the total number of operations to form the Fock matrix will range from about σN^4 down to approximately σN^2, if integral cutoffs are used, (see below). Assuming that this work is evenly distributed over the processors (p = the number of processors) and γ is the processor speed in DP floating point operations per second (FLOPS) the Fock matrix formation will require at minimum $\sigma N^2/p\gamma$ seconds. To pass the density and Fock matrix blocks around the loop of processors requires p successive sends of approximately N^2/p DP words of data between neighboring processors. If we assume the startup latency for communication is α seconds and the communication bandwidth between nodes is β DP words per second, the total communication time will be $p\alpha + N^2/\beta$. This leads to a worst-case computation/communication ratio of $\sigma N^2\beta/[\gamma p(p\alpha + N^2)]$. Assuming $\gamma \approx \beta$ and α is small, communication will start to dominate this step when $\sigma N^2 < pN^2$. Using our current two electron integral algorithm, σ is in the range of 100–1000 operations, so that the communication time is substantially less than the computation time. This ratio is improved by the fact that the communication and computation is overlapped, so that processors are not idle during the communication of the next set of shell blocks.

This algorithm requires storage of N^2/p elements of the density and Fock matrices per node. Additionally, an equal amount of message buffer space is required. For the problems we have studied (<350 basis functions) these required less than 32 Kbytes per node, an insignificant fraction of the available memory on any of the parallel computers used (see Appendix).

Since the two electron integrals do not change during the SCF calculation a choice must be made whether to precalculate and store the integral list or to recalculate them in each iteration – this latter option is the "direct" SCF method. Due to the limited memory and lack of efficient parallel I/O, storing the integral list is impractical on massively parallel computers. Nevertheless, while precalculating the integrals might seem to save much computational effort, there are two reasons why direct methods are efficient even on serial computers. The first is that the integral lists are so large that they must be stored on magnetic disk, so the retrieval time can be a significant fraction of the time required to recalculate the integral. More importantly, as will be described below, there are fast methods for determining in each iteration which integral blocks need not be calculated because they will not have significant contributions. A detailed analysis of the algorithms necessary to perform SCF calculations on very large systems has been performed by Panas et al. [8].

The integral estimation scheme implemented in MPSCF is that described by Häser and Ahlrichs [5]. This method exploits the relation:

$$|(\mu\nu \mid \lambda\sigma)| \leqslant (\mu\nu \mid \mu\nu)^{1/2}(\lambda\sigma \mid \lambda\sigma)^{1/2}$$

Thus, to form the upper limit of any integral the matrix:

$$Q_{\mu\nu} = (\mu\nu \mid \mu\nu)^{1/2}$$

is required. Actually since the integrals are computed by shell blocks Q need only be nblocks × nblocks. The Q matrix is precalculated in parallel and matrix element $Q_{\mu\nu}$ is distributed to all processors "responsible" for integrals $(\mu\nu/\lambda\sigma)$. During each iteration, the upper bound of an integral sub-block is multiplied by the corresponding element of ΔP and compared to a threshold (typically 10^{-10}). If this value is below the threshold then the integral block is excluded from contributing to the Fock matrix. In order to determine the effect of such cutoffs on large direct SCF calculations, we ran a variety of different test calculations using Gaussian 88 (which uses a simple integral cutoff algorithm) on our Cray X-MP. The exponent of the *overall* computational complexity of the direct SCF was determined using the relation:

$$\text{Exponent} = \frac{\log\left(\dfrac{time}{time_{\text{ref}}}\right)}{\log\left(\dfrac{nbf}{nbf_{\text{ref}}}\right)}$$

In this expression "*time*" is the total CPU time for the direct SCF calculation on a problem whose number of basis functions equals "*nbf*". Similarly "$time_{\text{ref}}$" is the time for the reference calculation which has "nbf_{ref}" basis functions. A medium sized, 133 basis function test case was used as a reference rather than a very small test case in order to reduce the effect of overhead (such as reading the input) on the timing results. The results, shown in Fig. 3, indicate that the cutoffs can dramatically reduce the computational complexity of the direct SCF from $O(N^4)$ to $O(N^{2.5})$, approaching the predicted asymptotic quadratic complexity [8]. Note that the exact values of these exponents are somewhat dependent on both the detail of the test calculations (e.g. the type of basis set and the shape of the molecules) and the size of the reference compound.

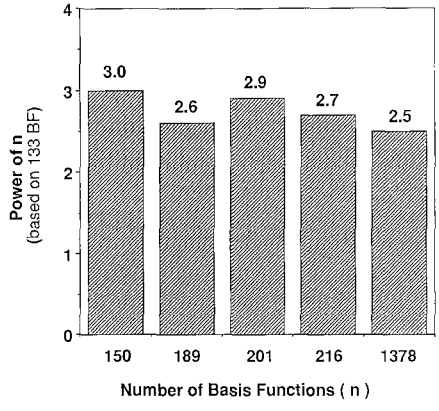

Fig. 3. Basis function power dependence for complete direct SCF calculations using Gaussian 88 running on a Cray X-MP

As initially implemented, MPSCF used the *sp* block integral subroutine "SHELL" from Gaussian 80 [9, 10] and was limited to only *s* and *p* type basis functions. We are currently implementing a much more sophisticated integral algorithm outlined by Head-Gordon and Pople [11]. This involves constructing each set of contracted integrals $(ab \mid cd)$ by first computing $(e0 \mid f0)$ where *a*, *b*, *c* and *d* give the angular momentum on each center and $e = a + b, f = c + d$. The integrals $(e0 \mid f0)$ are formed by contracting auxiliary integrals for each set of primitive quartets which are computed using the vertical recursion relation of Obara and Saika [12]. The horizontal recursion relation of Head-Gordon and Pople is then used to shift the angular momentum of the $(e0 \mid f0)$ integrals until the target integrals are formed. This algorithm will allow the inclusion of basis functions with arbitrarily high angular momentum.

The results in Fig. 3 indicate that for large chemical systems (above a few hundred basis functions) the integral estimation schemes mentioned above decrease the computational complexity of the Fock matrix formation from $O(N^4)$ down to $O(N^{2.5})$. Hence for chemical systems of the size we are interested in studying, the Fock matrix formation will have an asymptotic complexity that is *lower* than the matrix operations which are $O(N^3)$. Of course the multiplier of these complexities is different; approximately 100 for the Fock matrix formation (see analysis above) and 10 for the matrix manipulations. Nevertheless, for large problem sizes the matrix manipulations will start to dominate the SCF computation even on serial computers. In a parallel direct SCF program with the distributed Fock matrix formation and serial matrix manipulations, the problem is even more acute. Hence in order to use efficiently even a modest number of processors it is important to parallelize the $O(N^3)$ matrix manipulations.

Using our distributed matrix library (implementation described in next section) we have parallelized all of the matrix operations (except for the disk I/O which is inherently serial on our nCUBE 2). At present this library supports matrices distributed by sub-blocks and by columns. The two most common (and time consuming) matrix operations in the SCF program are multiplication and diagonalization. The matrix multiplication is implemented for both the column and block-distributed matrices. However a "direct" implementation of the matrix multiplication for the block-distributed matrix requires a large amount of communication since to perform $\boldsymbol{C} = \boldsymbol{A} * \boldsymbol{B}$, the product $(A_{ik}B_{kj})$ must be formed for every combination of sub-blocks. To minimize the communication required to form these products, the matrices *A* and *B* are converted from block-distributed to column-distributed format. Then, the columns of the matrix *A* are shipped around the loop of processors combining with locally held columns of *B* to form *C*. After the columns have been transmitted completely around this loop,

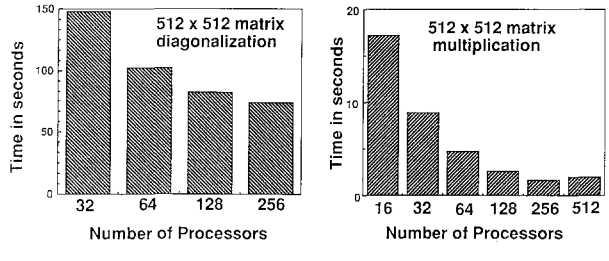

Fig. 4. Parallel timings for matrix operations on the nCUBE

(requiring # processor steps), the column-distributed C matrix has been completed. Timing results for a 512×512 matrix multiply are shown in Fig. 4.

Using the definitions for p, N, α, β, and γ defined in the analysis of the Fock matrix formation, the total time required for the floating point operators is $2N^3/p\gamma$ seconds. The total time required to pass a block containing a column of N values around a loop N times is $N(\alpha + N/\beta)$, leading to a computation-communication ratio of $2N^2/[p\gamma(\alpha + N/\beta)]$. The matrix multiply requires storing at most N^2/p matrix elements on each node (with a minimum of one column from each matrix in the case where $p > N$).

The parallel matrix diagonalization is based on an early version of Cleve Moler's Eiscube routines. Basically, this is a column-distributed Householder tridiagonalization followed by QR iteration. The timing results for this procedure are also given in Fig. 4. The analysis is somewhat more complex than for the matrix multiply. The total floating point operations required are $9N^3$, giving a total time of $9N^3/p\gamma$. The matrix diagonalization requires a combination of global broadcasts and reductions. This was implemented using the hypercube connectivity to yield a total communication cost of $3N(\alpha + N/2\beta) \log_2(p)$ and a computation-communication ratio of $3N^2/[p\gamma(\alpha + N/2\beta) \log_2(p)]$. The diagonalization must store the original matrix, its eigenvectors and the tridiagonal reduction space, yielding a total memory requirement of $3N^2/2$ words of memory.

3 Implementation

Implementation style has traditionally been a minor consideration in the development of quantum chemistry programs. The emphasis on program efficiency and the relatively awkward structuring and memory allocation features of FORTRAN have conspired to create quantum chemistry programs built of specialized subroutines that were opaque and difficult to re-use. Similarly, since individual quantum chemistry programs are usually used by small groups of users on a narrow class of computers, usable documentation of the internal workings of the programs is rare. This is particularly true for documentation of how the data is organized within the programs. High-level data constructs such as the wave function or basis set have components scattered across common blocks, inhibiting code re-use and intelligibility. Despite these difficulties, quantum chemistry has thrived; however, the growing complexity of the algorithms and the user interface, as well as the need to port these codes to complex computer architectures will require an increased emphasis on implementation issues within the community.

As an experiment we have implemented MPSCF using modern programming methods intended to increase code portability and re-use. MPSCF is written in the "C" programming language using a system for embedded code documentation called WEB [13]. Additionally, MPSCF uses object-oriented programming methodology [14] (although it's not written in a formally object-oriented language such as C++).

Our choice of the C programming language was primarily motivated by the availability of user-defined data types and standardized memory allocation and de-allocation*. The ability to create "aggregate" data types allows the

* Other advantages of C include: standardized file I/O, compatibility with UNIX system libraries, and numerous commercial and public-domain programming tools.

programmer to more naturally organize high-order data objects. For example, the specification for a basis function data type could include in a single entity all information associated with a particular basis function. This greatly simplifies the organization of data within a program and leads to enhanced code reusability.

Additionally, we are going beyond the C language capability to describe aggregate data types by using a language we have developed to describe both data types and operations on that data type (analogous to classes in C++ without the rigorous protection of private data). Minor extensions to the C syntax have been implemented to allow the automatic generation of subprograms to perform many possible operations on complex aggregate data types. These operations include reading and writing to many file formats, initializing the data types, releasing the memory allocated for the data types, and sending and receiving the data types between nodes of a parallel computer. Since the generation of these routines is completely automatic it is only necessary to modify the type description file and all of the routines for manipulating the type are updated. This greatly reduces the work associated with implementing and modifying complex data types. Currently, the basis set and atomic center information is handled using this facility.

The ability in C to allocate and free arbitrary chunks of memory also enhances code reusability since temporary storage can be allocated locally, simplifying the interface between different portions of programs. Such local memory allocation also allows memory to be used more efficiently which is particularly important on massively parallel computers where memory is relatively limited.

In the WEB language, the program documentation and source are combined. The program file (a ".web" file) can be run through preprocessors to produce either a TeX document or a C-language source file (see Fig. 5). In addition to simple text formatting, the WEB system provides a detailed index to the variables and subroutines as well as a sophisticated "macro" facility to allow cleanly written code without excessive numbers of subroutines. Since it is written in WEB, the MPSCF program can be printed out as a readable "book" where each logical

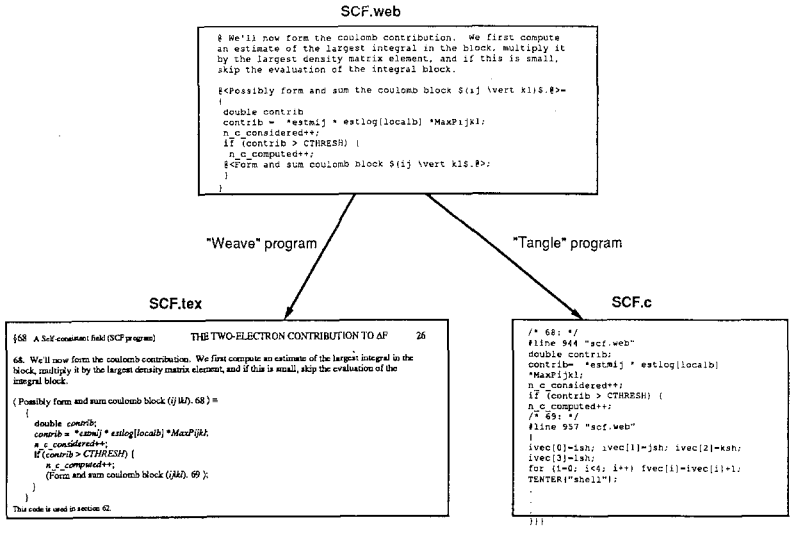

Fig. 5. Example of SCF.web code and c-source and TeX files generated using WEB programming tools

segment of the program is a separate chapter. This book contains a table of contents to the major program segments and an index of the variables and program sub-segments.

The MPSCF program is built on top of an object-oriented distributed matrix library. This library is written in C which is not formally an object-oriented language; however, with some effort, object-oriented methodologies can be used when coding in C. The key concept behind the object-oriented methodology is to encapsulate the high-level data structures and the associated functions. Experience on a wide variety of software applications has shown that this methodology produces particularly robust and re-usable code [15].

We implemented all of the linear algebra operations requried by the direct SCF algorithm in the distributed matrix library. Matrices are declared with a specific size and parallel distribution scheme and all matrix operations are carried out using functions within the library. The distribution of the matrix and the details of the function implementation are transparent to the user. For example, the transformation and diagonalization of the Fock matrix would include the following code:

```
{
    matrix s_half = dmt_create("S**(1/2)",nbasis,SCATTERED);
    matrix fock = dmt_create("Fock matrix",nbasis,SCATTERED);
    matrix tmp = dmt_create("Temp matrix",nbasis,COLUMNS);
    matrix fock_trans = dm_create("Trans.F matrix",nbasis,COLUMNS);
    matrix tmp = dmt_create("Temp matrix",nbasis,COLUMNS);
    /* Form s_half and fock (code not shown) */
    dmt_mult(s_half,fock,tmp);
    dmt_mult(tmp,s_half,fock_trans);
    dmt_diag(fock_trans,eig_vect,eig_val);
    dmt_write("scf_vect.dat",eig_vect);
    dmt_free(tmp);
}
```

In the example, we first create four matrices using the dmt_create routine. This associates a label, size, and distribution scheme with each of the matrices and allocates memory on each node. After the Fock and $S^{-1/2}$ matrices have been created (code not shown), the dmt_mult routine is used to transform the Fock matrix and dmt_diag is used to diagonalize the result. Finally, the eigenvector is written out to disk and the memory allocated for the temporary matrix, tmp, is released.

4 Results and discussion

Parallel results traditionally have been presented solely in terms of speedup curves (i.e. comparing an algorithm running on many processors to the same algorithm running on one processor). Unfortunately these can be misleading for a number of reasons. The main problem is that the best parallel algorithms are rarely the best for serial computers. Hence the reference "single-processor" timings on which the speedup results are based are usually biased in favor of the parallel implementation. A better performance measure is to compare the parallel implementation to a state-of-the-art serial implementation running on an appropriate serial or vector computer. (Speedup results can augment these

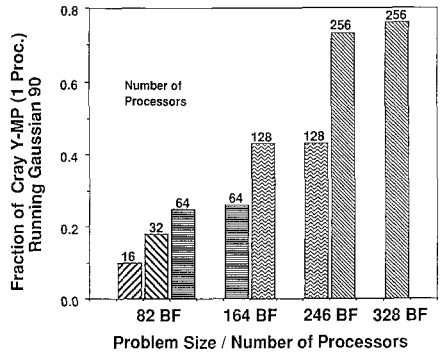

Fig. 6. Timing results for parallel direct SCF running in nCUBE 2

performance comparisons to indicate how the parallel implementation will scale up to larger parallel computers.)

For this reason we compared the performance of MPSCF with Gaussian 90 [16] running on a Cray Y-MP 8/864. The benchmark problems were a series of cytosine polymers run with a 3-21G basis set in C_1 symmetry yielding test problems with 82, 164, 246, and 328 basis functions. Figure 6 shows the results for the test problems on different numbers of processors. Since the matrix diagonalization is efficient only when the number of basis functions is greater than the number of processors, we were limited to 256 processors (1/4 of the nCUBE 2) for even the largest of the test cases.

The results were very encouraging. For systems over a few hundred basis functions, MPSCF running on 256 nCUBE processors performs nearly as well as Gaussian 90 running on a single processor Cray Y-MP. Hence, for large problems, the current version of MPSCF running on the full nCUBE should give roughly four times the through-put of Gaussian 90 running on a Cray Y-MP (since Gaussian can only use one processor at this time). Since the estimated peak performance of a single Y-MP processor is nearly that of 256 nCUBE 2 nodes, these results indicate that MPSCF has a net efficiency on the nCUBE comparable to the efficiency of Gaussian 90 on the CRAY (but both are well below the peak efficiency). This is significant because new parallel computers are considerably more cost effective in terms of actual cost per unit of floating point performance than vector supercomputers. Hence, for large-scale SCF calculations parallel computers should be considerably more cost effective than vector supercomputers.

To get estimates of the asymptotic performance of the MPSCF program we ran two speedup test sequences. These curves along with the ideal speedup line are shown in Fig. 7. Due to the limited memory on each node, these speedups were calculated from relatively small test cases. The lower curve is the speedup for the 82 basis function cytosine monomer calculated relative to the timing on a single processor. The middle curve is the speedup for the 164 basis function cytosine dimer calculated relative to the timing on four processors (the smallest number having enough total memory to run the problem). For both of these test cases the parallel efficiency rapidly drops off as the number of processors gets greater than about half the number of basis functions. The reasons for this drop in efficiency are shown in Fig. 8. This figure compares the timings for the Fock matrix formation and the matrix manipulations for the 82 basis function test case running on different numbers of nCUBE 2 processors. The Fock matrix formation is very efficient, even on 128 processors the parallel efficiency is greater than 60%. In contrast, the matrix manipulation shows no speedup for more than

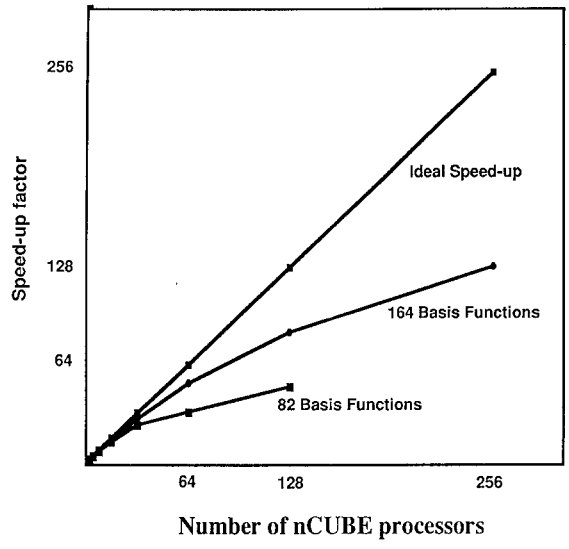

Fig. 7. Speed-up curves for direct SCF on the nCUBE 2 computer for problems of size 82 and 164 basis functions

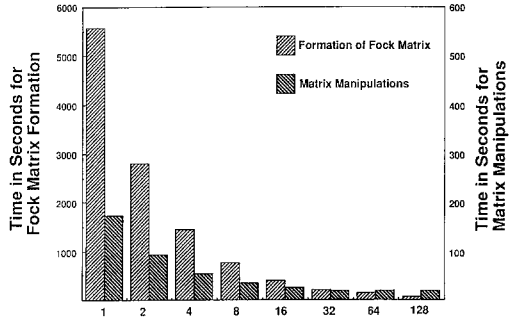

Fig. 8. Comparison of parallel timings for components of direct SCF calculation for a problem with 82 basis functions

16 processors and on 128 processors is requiring almost half as much time as the Fock matrix formation (as compared to 3% on a single processor). This indicates that the column distribution used in the matrix manipulations will limit the performance of the parallel direct SCF unless the number of basis functions is much greater than the number of processors being used.

5 Conclusions and future work

The results presented here show that for some applications, parallel computers are a cost-effective alternative to vector supercomputers. However, the next generations of parallel computers will have a much more significant impact since they will provide the capability to perform quantum chemical calculations that are at present unfeasible. Although predictions of future computational needs and capabilities are notoriously inaccurate [17] they provide a useful measure of future successes. Figure 9 is a projection of single point direct SCF capabilities on a wide range of serial and parallel computers. This projection was made assuming typical biochemical composition using 6-31G** basis sets. We assumed

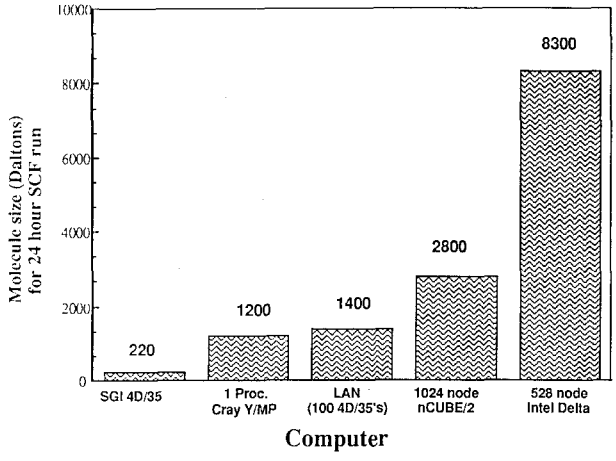

Fig. 9. Projected performance of current and future computers in terms of molecular weight of compounds for which a 6-31G** single point SCF calculation can be performed in a single day of CPU time

that the SCF computation time would scale as $N^{2.5}$ and included very crude estimates of the communication overhead and processor efficiency. This plot contains several interesting results. One is that local area networks of high-end workstations will prove to be impressive computational resources, comparable to single processors of vector supercomputers. However, the relatively low bandwidth of conventional LAN's (e.g. ethernet) will not allow such networks to supplant the role of parallel supercomputers. Another exciting implication of these projections is that with the arrival of the Touchstone Delta, parallel computers have sufficient power to allow SCF calculations on chemical systems large enough to truly interest researchers in medicinal chemistry and materials science.

We plan eventually to extend the parallel quantum chemistry package to include electron correlation methods such as coupled cluster and configuration interaction. However, the difficulty of efficiently parallelizing these extremely complex algorithms is overwhelming for a single research team. It is our hope that the development of new programming tools and methodologies, such as those described in this paper, will lead to more cooperative program development within the quantum chemistry community. Such community synergism, combined with the arrival of powerful new computer architectures, will guarantee that quantum chemistry will play an important role in the exciting new fields of chemical research.

Appendix

The performance characteristics of parallel computers are constantly changing, but to inform the reader of the current state-of-the-art, the following table lists the vital statistics for the nCUBE/2 and Intel IPSC/i860 installed and Sandia National Laboratories and the Intel Touchstone Delta installed at the Concurrent Supercomputing Consortium at the California Institute of Technology. The CPU speeds and interprocessor communication rates are the estimates provided by the manufacturers and should be interpreted as the "upper limits" on actual capabilities. The estimated processor speeds, theoretical peak performances, and DP Linpack benchmark results are from Dongarra [18]. The interprocessor communication rates for the nCUBE and Intel i860 are our measured interprocessor bandwidths. The bandwidth for the Intel Delta are Intel's predicted peak rates.

Computer	# Processors	Estimated processor speed (MFLOPS)	Memory per processor	Interprocessor communication bandwidth (Mbytes/sec)	Theoretical peak performance (MFLOPS)	Performance on DP Linpack benchmark (MFLOPS)
nCUBE/2 (Sandia)	1024	2.4	4 Mbytes	1.5 Mbytes/sec	2400	1900
Intel i860 (Sandia)	64	40	8 Mbytes /32 Mbytes [19]	2.1 Mbytes/sec	2600	1400
Intel Delta (Caltech)	512	40	16 Mbytes	200 Mbytes/sec	20000	13900

Acknowledgments. This work was carried out at Sandia National Laboratory under contract by the U.S. Department of Energy and supported by its Division of Basic Energy Sciences.

References

1. Clementi E, Corongui G, Detrich JM, Domingo L, Laaksonen A, Nguyen HL, Chin S (1984) Tech Report No POK-40, IBM
2. Colvin ME, Whiteside RA, Schaefer HF III (1989) in: Wilson S (ed) Methods in quantum chemistry Vol 3. Plenum, NY, p 167–237
3. Parallel Computing in Chemical Physics (1991) Org Theoret Chem Group, Argonne Natl Lab, July 17–19, 1991. 700 S Cass Ave, Argonne, IL 60439
4. MIMD parallel computers consist of independent, interconnected CPU's communicating via wires (message-passing) or a common memory bank (shared-memory). Shared memory MIMD computers are easier to program than message-passing computers, but at this time cannot efficiently interconnect large numbers of processors. Conversely, a message-passed architecture can be readily emulated by a shared-memory computer. An alternative to MIMD architectures is single-instruction multiple-data (SIMD) in which all processors work in lock-step, performing identical instructions on different data streams
5. Häser M, Ahlrichs R (1989) J Comput Chem 10(1):104
6. Pulay P (1982) J Comput Chem 3(4):556
7. Garey MR, Johnson DS (1979) Computers and intractability. A guide to the theory of NP completeness. Freeman, NY
8. Panas I, Almlöf J, Feyereisen MW (1991) Int J Quant Chem 40:797
9. Program #446, Quantum Chem Program Exchange, Indiana Univ, Bloomington
10. Pople JA, Hehre WJ (1978) J Comp Phys 77(20):161
11. Head-Gordon M, Pople J (1988) 89(9):5777
12. Obara S, Saika A (1986) J Chem Phys 84(7):3963
13. Knuth DE (1984) The Computer J 27(2):93
14. Pascoe GA (1986) Byte 11(8):139
15. Meyer B (1987) IEEE Software 4(2):50
16. Gaussian 90, Rev I (1990) Frisch MJ, Head-Gordon M, Trucks GW, Foresman JB, Schlegel HB, Raghavachari K, Robb M, Binkley JS, Gonzalez C, Defrees DJ, Fox DJ, Whiteside RA, Seeger R, Melius CF, Baker J, Martin RL, Kahn LR, Stewart JJP, Topiol S, Pople JA, Gaussian Inc, Pittsburgh PA
17. Early computer pundits believed there was little need for high-speed computers since only a limited number of mathematical tables were needed by the scientific community. (Described in: Von Neuman J (1946) transcript of talk given at the Princeton Institute for Advanced Study, May 15, 1946. Office of Naval Research, Wash DC)
18. Dongarra JJ (1992) Supercomputing Rev 5(3):54
19. 32 Processors have 32 Mbytes and the remaining 32 processors have 8 Mbytes