Theor Chem Acc (1998) 99:220–223
DOI 10.1007/s0021498000m6

Theoretical
Chemistry Accounts
© Springer-Verlag 1998

Regular article

Parallel implementation of a divide and conquer semiempirical algorithm

James J. Vincent, Steven L. Dixon, Kenneth M. Merz, Jr.

Department of Chemistry, The Pennsylvania State University, University Park, PA 16802, USA

Received: 5 December 1997 / Accepted: 13 February 1998 / Published online: 17 June 1998

Abstract. We have implemented a parallel version of the semiempirical divide and conquer program DivCon previously developed in our laboratory. By utilizing a parallel machine we are able to leverage the linear scaling of the divide and conquer algorithm itself to perform semiempirical calculations on large bio-molecules. The utility of the implementation is demonstrated with a partial geometry optimization of hen egg white lysozyme in the gas phase.

Keywords: Parallel divide and conquer – Parallel SCF

1 Introduction

Recently in our laboratory we implemented a divide and conquer algorithm for semiempirical calculations in a program called DivCon [1, 2]. This algorithm is based on a variation of the density matrix divide and conquer implementation of Yang and Lee [3]. Yang and coworkers have also implemented a semiempirical divide and conquer within the MOPAC program [4–6]. This method overcomes the N^3 time scaling of typical semiempirical calculations by breaking a system into smaller subsystems and solving a set of localized equations for the subsystem. Using this algorithm, semiempirical SCF calculations on large molecules can be carried out with normal workstations in a reasonable amount of time. Other approaches to linear scaling semiempirical calculations, such as those of Stewart [7] and Daniels et al. [8], have also been developed.

It is our goal in this work to leverage the linear scaling ability of the density matrix divide and conquer algorithm by implementing it in parallel. By doing so we hope to reduce the time needed for semiempirical calculations to a point where it is feasible to perform molecular dynamics using a fully semiempirical Hamiltonian.

The parallel implementation has already been suggested and outlined in Dixon and Merz [1]. Initial results were reported there also. We have completed the work and give full details of the implementation. Initial results of a geometry optimization of hen egg white lysozyme are presented to demonstrate the utility of the parallel implementation.

Correspondence to: K.M. Merz, Jr.

2 Method

The partitioning scheme of the divide and conquer algorithm lends itself naturally to a parallel implementation. In the divide and conquer algorithm the entire system is split into many smaller subsystems. It is this partitioning that allows the algorithm to overcome the N^3 scaling typically associated with semiempirical calculations requiring diagonalization of the Fock matrix. This same partitioning is used to implement the algorithm in parallel. Individual subsystems are distributed across processors in a parallel machine. In this manner the scaling of performance on a parallel machine is limited only by the time required to diagonalize the Fock matrix of the largest subsystem. Examples of this behavior are given in the results section.

A replicated data strategy was used for the parallel implementation. We prefer this method because of its simplicity and ease of implementation. Although replicated data implementations of parallel programs normally do not scale very well to large numbers of processors we find this is not a hindrance since the majority of our work is accomplished on a small number of processors, typically 64 or less. Currently it is much easier to obtain a small number of processors at various supercomputing sites than an entire machine of say 512 processors. Because of this we try to optimize our codes to run on the smaller number of nodes. In doing so we are willing to make a tradeoff in scaling performance for ease of implementation and maintenance. This allows us to make modifications and test new features more easily, while still getting excellent performance in the way we normally use the program. Just over 100 lines of additional FORTRAN code were needed for the entire parallel implementation, including robust error checking.

A more important limitation of the replicated data strategy is available memory. Each processor must instantiate all variables of the entire program. This limits the size of systems that can be used to the size that will fit on a single processor. The serial implementation of this algorithm is very memory efficient and can accommodate several thousand atom systems on a workstation with only 32 Mb of memory. The cutoff employed in the divide and conquer algorithm linearizes the storage requirements. Based on these observations we anticipate being able to carry out calculations on systems of over 10 000 atoms on most parallel machines. This exceeds our current needs. Memory requirements can be significantly reduced by implementing some portions of the code in a data parallel manner while still using the replicated data framework. Improvements in this area will be implemented in future versions of the DivCon program as the need arises.

The main features of the parallel implementation are typical of any replicated data program [9]. All processors read data from a file at startup, initialize a variety of global variables locally, and execute all serial regions of the code redundantly. One processor is named the master node and is responsible for all output. Communication consists mainly of global sum operations. Several operations, such as finding the global minimum and maximum of an array, are carried out very efficiently in parallel directly in a global communication operation. The message passing interface (MPI) was used for all communications [10]. The parallel version runs on any machine with an MPI implementation available, including networked workstations.

Two portions of the code were implemented in parallel; building the Fock matrix, and diagonalizing the Fock matrix. Creating the Fock matrix does not account for significant time, relative to diagonalization, in standard semiempirical calculations. In the divide and conquer implementation diagonalization time has been reduced such that creating the Fock matrix can require a significant percentage of the total computation time. Together these two portions of the code account for 99% of total computation time on a serial machine. Note that even with a perfect parallel implementation we can only expect a maximum speedup of 100-fold.

The Fock matrix is created in parallel by distributing atoms among processors in a cyclic fashion. No extra load balancing is performed during this process. Each processor is responsible for all matrix elements associated with a subset of atoms. The subset of atoms assigned to a processor is not related to the subsystem scheme employed by the divide and conquer algorithm. Since we are using a replicated data strategy, every processor has equal access to the full density matrix. A single communication scatters the entire matrix to all processors.

Diagonalization of the Fock matrix is performed by solving a localized set of Roothan-Hall equations for each subsystem. Distribution of this work by subsystems across processors allows a larger granularity while still enabling effective load balancing. In other parallel implementations of standard SCF calculations, diagonalization of the entire Fock matrix is carried out in parallel [11–14]. Much research has been devoted to general methods for parallel matrix diagonalizations due to their necessity in many scientific programs [9]. In our implementation we avoid a parallel diagonalization altogether by breaking the work up across processors at a level above the diagonalization itself. Each processor completes a diagonalization of the local Fock matrix for one or more subsystems. An advantage of this method is that we can exploit the superior performance of native serial diagonalization routines on many machines. The disadvantage is that parallel speedup is limited by the single largest subsystem. This effect is shown in the results section.

Load balancing of subsystems across processors is carried out with a simple Greedy algorithm that uses the number of orbitals in a subsystem and completion time per subsystem to balance the work load effectively across processors [15]. The algorithm treats subsystems as units of work of varying size and distributes them across processors in such a way that each processor has an equal share of the total work. The optimal load balance achieved is normally not a perfect balance since most often the work cannot be divided exactly among processors. In the limiting case a single processor would be assigned one subsystem, the largest subsystem in the calculation. A simplified pseudo code algorithm is given below.

```
Greedy load balancing:
sort subsystems in decreasing order
determine target solution
for i = 1, number_subsystems
        for j = 1, number_pe's
                if ( FEASIBLE(my_solution,i) ) then
                    my_solution = UNION(my_solution, i)
                endif
        end
end
```

The first step of the algorithm sorts a list of all subsystems by size, using either the number of orbitals in that subsystem, or the completion time for diagonalization of the local Fock matrix. In the second step a target solution based on the measurement used in the first step is computed. This is simply the total work to be performed divided by the number of processors. For instance, if the sum of completion times of all subsystems is 1000 s, then on 8 processors the target would be 1000/8. This is how much work each processor should ideally be responsible for. In the loop that follows, subsystems are simply assigned to processors with the condition that each processor must not exceed the target value when a subsystem is assigned to it. The FEASIBLE routine checks this condition. If FEASIBLE returns true the subsystem is added to that processors pool of subsystems (UNION (my_solution, I)). If not the next processor is checked. If all processors exceed the target value the subsystem is assigned to the processor with the lowest work total at that point.

The first iteration of an SCF cycle uses the number of orbitals in a subsystem for sorting and as the selection criterion in the FEASIBLE routine. The second iteration uses actual completion times. Subsystem distribution among processors is then left in this balanced state until a new SCF cycle begins. Load balancing is performed once in a single point calculation, and once at the start of each new SCF cycle in a geometry optimization. During geometry optimization subsystems can be regenerated. Any new distribution of atoms among subsystems requires load balancing.

3 Results

We present performance results for several systems. Timings for several systems are presented in Table 1. Speedup as a function of the number of processors for this data is given in Fig. 1. The limitation of the implementation is clear in this graph. Bovine pancreatic trypsin inhibitor (BPTI) shows near linear scaling until 16 processors. At that point the total speedup is limited by the size of the largest single subsystem. One processor is performing all calculations for the largest subsystem and no matter how many processors are used the entire calculation will not go faster. Because of this limitation we compute maximum speedup possible based on subsystem sizes within the program itself. This information can then be used to choose an optimal number of processors on a parallel machine.

The effectiveness of the Greedy load balancing algorithm is shown in the three data sets in Fig. 2. Each bargraph set shows the total completion time for all work for each of eight processors. The first set shows completion times without load balancing. This demonstrates

Table 1 Completion times for single SCF cycles of several systems. All timing values are wall times in s on a Cray T3 E for the first (two pass) SCF cycle using PM3. See [1] for two pass method.

Processors	Water Box[a]	BPTI[b]	Glycine[c]	Lysozyme[d]
1	837.28	10395.78	215.04	25109.32
2	421.52	5219.78	109.62	12597.23
4	214.37	2628.55	57.75	6352.96
8	110.68	1343.08	31.99	3224.80
16	59.16	742.07	19.37	1781.83
32	33.80	730.86	13.65	1035.86

[a] Water box: 216 waters in a rectilinear box, 648 atoms
[b] BPTI: bovine pancreatic trypsin inhibitor, 892 atoms
[c] Glycine: linear chain of 100 glycines, 703 atoms
[d] Lysozyme: hen egg white lysozyme, 1960 atoms

the need for balancing and clearly shows performance gains that can be made. The second set shows the completion times for each processor using a cyclic load balancer. In this algorithm subsystems are distributed to processors in cyclic fashion until all subsystems have been assigned to a processor. The balance achieved is clearly better than no load balance, but there is still room for improvement. Figure 3 shows completion times across processors for the Greedy load balancing algorithm. This provides the best results by giving the optimal balancing solution for a set of subsystems and processors. Note that in some instances the optimal solution will not necessarily be a perfectly balanced solution. This behavior is evident in the timing data for BPTI presented in Fig. 1. In this instance one processor is assigned a single large subsystem that requires more time than any of the other processors with their combined subsystems. Since subsystems are not broken down further this is the optimal solution.

A breakdown of times spent in various regions of the code as a function of the number of processors is presented in Fig. 3. Even though we are using a replicated data implementation that requires very large global communication operations it is clear from this graph that we are not limited by communication time. On 32 nodes of a Cray T3E diagonalization of individual subsystems still accounts for 74% of total wall time. Though serial regions of the code and communications are not the limiting factor on 32 nodes, they are significant enough to show that this implementation will not scale well to very large numbers of processors.

4 Application

To demonstrate the utility of the parallel implementation we performed a geometry optimization of hen egg white lysozyme, a protein with 1960 atoms, using 4963 basis functions. The protein was broken into 129 subsystems for the divide and conquer algorithm, giving a maximum speedup of 26 based on the single largest subsystem. Actual measured speedup for this system was 24 on 32 nodes of a Cray T3E. A total of 1023 geometry optimization steps, requiring 2111 SCF cycles, were completed. The gradient norm, shown in Fig. 4, was

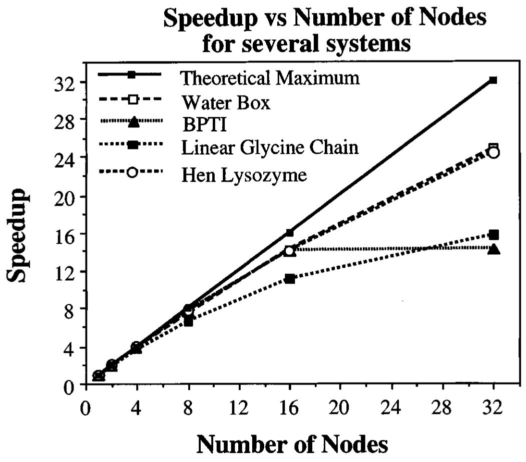

Fig. 1. Parallel performance of the DivCon program

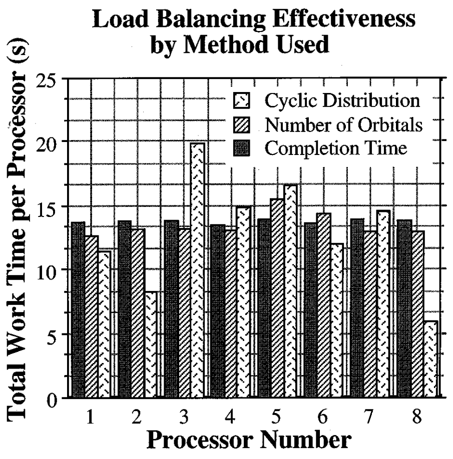

Fig. 2. Effectiveness of different load balancing schemes

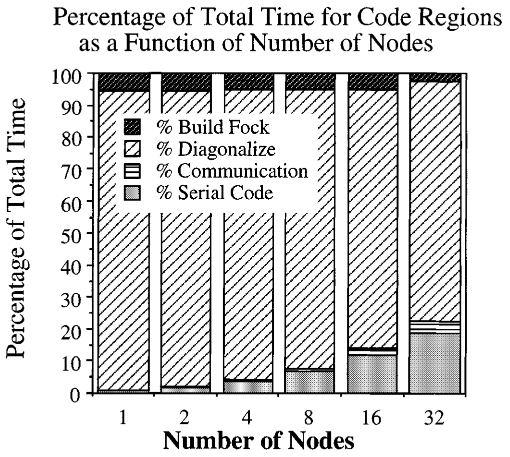

Fig. 3. Breakdown of computation time for each region of code as a function of processors used in parallel

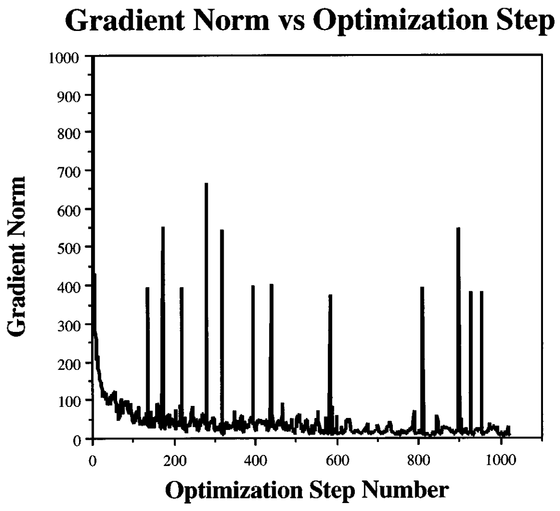

Fig. 4. Gradient norm as a function of optimization step number during geometry optimization of hen egg white lysozyme

reduced to 4.0 with an energy change between optimization steps of 0.0016 kcal. Converged SCF cycles averaged 528 s (wall time). Complete optimization required 13.4 days of CPU time on 32 nodes. To complete this optimization on a single processor workstation with the same performance as one T3 E node would require approximately 1 year of CPU time.

Conclusions

Our parallel implementation of the semiempirical divide and conquer algorithm allows us to leverage the serial linear scaling performance of the algorithm on parallel supercomputers. This in turn will allow us to carry out studies on large systems which were previously prohibitively expensive. We have demonstrated this ability with a geometry optimization of the protein, hen egg white lysozyme.

The performance of our algorithm is achieved through a balance of parallel optimization and maintainability. We have purposely chosen a very minimal implementation to enable easy modification and maintenance of the program. In most cases the implementation scales well on small numbers of nodes. We have demonstrated this with a 24-fold speedup on 32 nodes for the lysozyme.

Ongoing work in our laboratory includes the incorporation of periodic boundary conditions, particle mesh Ewald summation for long-range Coulombic interactions, and improved subsetting schemes into the divide and conquer program. Due to the simplicity of the parallel implementation these features and others can be developed without rewriting the parallel regions of the code. In addition, current work is under way to implement a coupled potential version of the molecular dynamics program ROAR [16] using DivCon.

Acknowledgments. This work was funded by the National Science Foundation Graduate Research Traineeship program for High Performance Computing. Computer time was granted through the National Science Foundation Meta-Centers.

References

1. Dixon SL, Merz KM Jr (1997) J Chem Phys 107:879
2. Dixon SL (1996) J Comp Chem 104:6643
3. Yang W, Lee TS (1995) J Chem Phys 103:5674
4. York DM, Lee TS, Yang W (1996) J Am Chem Soc 118:10940
5. York DM, Lee TS, Yang W (1996) Chem Phys Lett 263:297
6. Lee TS, York DM, Yang W (1996) J Chem Phys 105:2744
7. Stewart JJJP (1996) Int J Quantum Chem 58:133
8. Daniels AD, Millam JM, Scuseria GE (1997) J Chem Phys 107:425
9. Foster I (1995) Designing and building parallel programs: concepts and tools for parallel software engineering. Addison-Wesley, Reading, Mass.
10. MPI Forum (1993) University of Tennessee, Knoxville, Tenn.
11. Brode S, Horn H, Ehrig M, Moldrup D, Rice JE, Ahlrichs R (1993) J Comp Chem 14:1142
12. Feyereisen M, Kendall RA (1993) Theor Chim Acta 84:289
13. Pettersson LGM, Faxen T (1993) Theor Chim Acta 85:345
14. Furlani TR, King HF (1995) J Comp Chem 16:91
15. Horowitz E, Sahni S (1978) Fundamentals of computer algorithms. Computer Science Press, Polomac, MD
16. Merz KM Jr, Cheng A, Damodaran K, Stanton RV, Vincent JJ (1997) ROAR 1.0. Oxford Molecular, Oxford