METHODS IN COMPUTATIONAL CHEMISTRY

VOLUME 3

CONCURRENT COMPUTATION IN CHEMICAL CALCULATIONS

EDITED BY
STEPHEN WILSON

Methods in Computational Chemistry

Volume 3
Concurrent Computation in Chemical Calculations

METHODS IN COMPUTATIONAL CHEMISTRY

Volume 1 Electron Correlation in Atoms and Molecules
Edited by Stephen Wilson

Volume 2 Relativistic Effects in Atoms and Molecules
Edited by Stephen Wilson

Volume 3 Concurrent Computation in Chemical Calculations
Edited by Stephen Wilson

Methods in Computational Chemistry

Volume 3
Concurrent Computation in Chemical Calculations

Edited by

STEPHEN WILSON
Rutherford Appleton Laboratory
Oxfordshire, England

Springer Science+Business Media, LLC

ISBN 978-1-4615-7418-7 ISBN 978-1-4615-7416-3 (eBook)
DOI 10.1007/978-1-4615-7416-3

Originally published by Plenum Press , New York in 1989
Softcover reprint of the hardcover 1st edition 1989

Contributors

C. W. Bauschlicher, Jr., NASA Ames Research Center, Moffett Field, California 94035

M. E. Colvin, Lawrence Livermore National Laboratory, Livermore, California 94550

N. Koga, Institute for Molecular Science, Myodaiji, Okazaki 444, Japan

K. Morokuma, Institute for Molecular Science, Myodaiji, Okazaki 444, Japan

U. Nagashima, Institute for Molecular Science, Myodaiji, Okazaki 444, Japan

S. Obara, Department of Chemistry, Kyoto University, Kyoto, Japan

H. F. Schaefer III, University of Georgia, Athens, Georgia 30602

D. W. Schwenke, NASA Ames Research Center, Moffett Field, California 94035

P. R. Taylor, NASA Ames Research Center, Moffett Field, California 94035

R. A. Whiteside, Sandia National Laboratory, Livermore, California 94550

S. Wilson, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, U.K.

S. Yabushita, Department of Chemistry, Hiroshima University, Hiroshima, Japan

S. Yamamoto, Institute for Molecular Science, Myodaiji, Okazaki 444, Japan

From the Preface to Volume 1

Today the digital computer is a major tool of research in chemistry and the chemical sciences. However, although computers have been employed in chemical research since their very inception, it is only in the past ten or fifteen years that computational chemistry has emerged as a field of research in its own right. The computer has become an increasingly valuable source of chemical information, one that can complement and sometimes replace more traditional laboratory experiments. The computational approach to chemical problems can not only provide a route to information that is not available from laboratory experiments but can also afford additional insight into the problem being studied, and, as it is often more efficient than the alternative, the computational approach can be justified in terms of economics.

The applications of computers in chemistry are manifold. A broad overview of both the methods of computational chemistry and their applications in both the industrial research laboratory and the academic research environment is given in my book *Chemistry by Computer* (Plenum Press, 1986). Applications of the techniques of computational chemistry transcend the traditional divisions of chemistry—physical, inorganic, and organic—and include many neighboring areas in physics, biochemistry, and biology. Numerous applications have been reported in fields as diverse as solid-state physics and pesticide research, catalysis and pharmaceuticals, nuclear physics and forestry, interestellar chemistry and molecular biology, surface physics and molecular electronics. The range of applications continues to increase as research workers in chemistry and allied fields identify problems to which the methods of computational chemistry can be applied.

The techniques employed by the computational chemist depend on the size of the system being investigated, the property or range of properties

that are of interest, and the accuracy to which these properties must be measured. The methods of computational chemistry range from quantum mechanical studies of the electronic structure of small molecules to the determination of bulk properties by means of Monte Carlo or molecular dynamics simulations, from the study of protein structures using the methods of molecular mechanics to the investigation of simple molecular collisions, from expert systems for the design of synthetic routes in organic chemistry to the use of computer graphics techniques to investigate interactions between biological molecules.

The computers employed in chemical calculations vary enormously, from the small microcomputers used for data analysis to the large state-of-the-art machines that are frequently necessary for contemporary *ab initio* calculations of molecular electronic structure. Increasingly large mainframe computers are departing from the traditional von Neumann architecture with its emphasis on serial computation, and a similar change is already underway in smaller machines. With the advent of vector processing and parallel processing computers, the need to match an algorithm closely to the target machine has been recognized. Whereas different implementations of a given algorithm on traditional serial computers may lead to programs that differ in speed by a factor of about 2, factors of 20 were not uncommon with the first vector processors and larger factors can be expected in the future.

With the increasing use of computational techniques in chemistry, there is an obvious need to provide specialist reviews of methods and algorithms so as to enable the effective exploitation of the computing power available. This is the aim of the present series of volumes. Each volume will cover a particular area of research in computational chemistry and will provide a broad-ranging yet detailed analysis of contemporary theories, algorithms, and computational techniques. The series will be of interest to those whose research is concerned with the development of computational methods in chemistry. More importantly, it will provide an up-to-date summary of computational techniques for the chemist, atomic and molecular physicist, biochemist, and molecular biologist who wish to employ the methods to further their research programs. The series will also provide the graduate student with an easily accessible introduction to the field.

Preface

Recent years have seen the proliferation of new computer designs that employ parallel processing in one form or another in order to achieve maximum performance. Although the idea of improving the performance of computing machines by carrying out parts of the computation concurrently is not new (indeed, the concept was known to Babbage), such machines have, until fairly recently, been confined to a few specialist research laboratories. Nowadays, parallel computers are commercially available and they are finding a wide range of applications in chemical calculations.

The purpose of this volume is to review the impact that the advent of concurrent computation is already having, and is likely to have in the future, on chemical calculations. Although the potential of concurrent computation is still far from its full realization, it is already clear that it may turn out to be second in importance only to the introduction of the electronic digital computer itself.

In the first chapter in the present volume, I provide an overview of the general aspects of parallel computers and concurrent computation, emphasizing those aspects that are particularly relevant to chemical calculations. The spectrum of parallel computer architectures ranges between, on the one hand, those machines that contain a relatively small number of very powerful processors, and, on the other hand, machines with much larger numbers of less powerful processors. The machines manufactured and marketed by Cray Research, Inc., are typical of the first type, and in Chapter 2, Peter R. Taylor, Charles W. Bauschlicher, and David W. Schwenke provide a detailed description of the use of Cray supercomputers in chemical calculations. In Chapter 3, Keiji Morokuma and his colleagues assess the use of supercomputers manufactured in Japan in chemical calculations. In Chapter 4 we turn our attention to machines

containing a larger number of processors as Michael E. Colvin, Robert A. Whiteside, and Henry F. Schaefer III describe hypercube architectures and their use in calculations of chemical interest.

Machines capable of performing parts of a computation concurrently will almost certainly become increasingly important in chemical calculations over the next few years as chemists continue to extend their horizons to handle larger and increasingly complex systems. Together the four chapters in this volume provide a broad-ranging yet thorough analysis of the most important aspects of contemporary research into the problem of the effective exploitation of parallel computers in chemical studies.

Stephen Wilson

Wood End

Contents

3. Chemical Calculation on Japanese Supercomputers

K. Morokuma, U. Nagashima, S. Yamamoto, N. Koga, S. Obara, and S. Yabushita

4. Quantum Chemical Methods for Massively Parallel Computers

Michael E. Colvin, Robert A. Whiteside, and Henry F. Schaefer III

1

Parallel Computers and Concurrent Computation in the Chemical Sciences

STEPHEN WILSON

1. Introduction

Computers play a central role in the practical applications of the vast majority of the apparatus of theoretical chemistry. Indeed, the outlook for performing accurate quantum mechanical calculations of the electronic structure and properties of molecules changed radically in the early 1950s with the advent of the electronic digital computer. Since that time, steady progress has been made with both the theoretical apparatus and the computational techniques required to approximate the electronic structure of molecules, and we now find ourselves able not only to perform high-precision calculations for small molecules but also to compute the properties of quite large molecular systems. In more recent years, other areas of applications for computers in chemistry have arisen, for example, the simulation of liquids and solids using Monte Carlo and molecular dynamics methods, the study of complex reaction schemes, and the use of interactive molecular graphics in the study of biomolecular systems.

Since the first electronic computers were constructed, the speed of such machines has increased by many orders of magnitude. For example, the EDSAC1, which was completed in Cambridge in 1949, had an average performance of around 100 arithmetic operations per second, whereas

STEPHEN WILSON • Rutherford Appleton Laboratory, Chilton, Oxfordshire OX11 0QX, U.K.

the CRAY-1, which was first introduced in 1976 and an often quoted benchmark of computer performance, can carry out over 100 million arithmetic operations per second for some applications. The power of typical mainframe computers, measured in millions of instructions per second (MIPS), has increased exponentially with time over the past 25 years, as can be seen from Figure 1.

There are essentially two underlying reasons for this increase in speed. On the one hand, there has been a revolution in electronic engineering. The first computers used vacuum tubes, which were slow, bulky, and unreliable and consumed a lot of power. Their memories were small in capacity, large in physical size, and very slow. For example, the EDSAC1 used around 3000 thermionic values allowing a clock period of 2 μsec, and had a total fast memory of 512 words! The CRAY-1, on the other hand, uses emitter coupled logic technology with a clock period of 12.5 nsec, and a central memory of up to two million 64-bit words. However, the increase in speed coming from the improvements in electronics only accounts for a factor of 160, whereas the increase in arithmetic speed amounts to a factor of 10^6. The second and much more significant reason for the increase in arithmetic

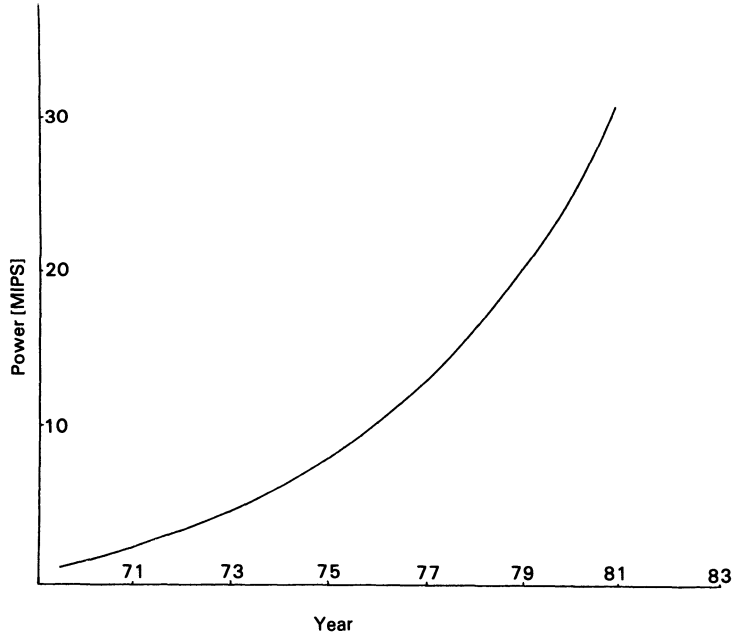


Figure 1. The power of typical mainframe computers, measured in millions of instructions per second (MIPS), as a function of time. The power available on typical mainframe computers has increased exponentially with time over the past twenty years. (Taken from Ref. 2.)

speed has come from improvements in the design of the architecture, and, in particular, by incorporating various forms of parallelism. Even the earliest computers employed some sort of parallelism: all the bits of a number could be added in parallel, the central processing unit could be performing operations while the card reader or magnetic drum was active, memory could be divided into separate banks which could each be accessed in parallel, and so on. The CRAY-1 was novel in that it included vector registers containing 64 words, and hardware instruction for operating on these vectors. It is a vector processor.

Over the past decade there has been a proliferation of new computer designs that employ parallel processing in one form or another in order to achieve their maximum performance. We now appear to be entering an era of massive computing power. No longer will computing power be the scarce resource it has been for the last few decades. However, to fully realize this power we must escape from the straightjacket imposed by the serial machines of the past; we must learn to compute concurrently. The old proverb "Many hands make light work" encapsulates the central thesis of parallel computing. Indeed, the idea of improving the performance of computing machines by carrying out parts of the computation concurrently

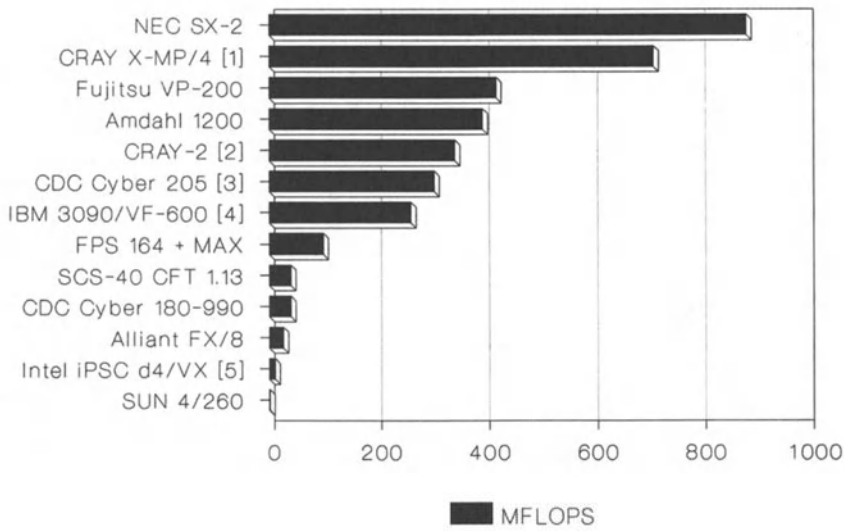


Figure 2. The rate of computation achieved in solving a system of equations on some contemporary computers. The rate of computation is measured in millions of floating point operations per second (MFLOPS). (Based on Ref. 14.)

is by no means new. The concept, in fact, predates the electronic computer. It was known to Charles Babbage, Lucasian Professor of Mathematics at Cambridge University 1828–1839, whose pioneering work on the "Application of Machinery to the Computation of Mathematical Tables" is often seen as the dawn of the computing age. Many mechanical and electromechanical calculators that were in use in the 1940s processed digits simultaneously. However, until fairly recently, parallel electronic computing machines have been confined to a few specialist research laboratories. Over recent years, parallel computers have become commercially available, and they are finding a wide range of applications in scientific calculations. In Figure 2, we provide an illustration of the wide range of machines available by comparing the rate of computation achieved in the solution of a system of equations of order 1000.

This volume provides a review of the impact that concurrent computation has had, and is continuing to have, on calculations in the chemical sciences. The potential of this advance is still far from its full realization, not least because the development of computer software is often found to lag about five years behind the developments in hardware. However, it is already clear that the advent of concurrent computation on the computational sciences, in general, and chemical calculations, in particular, will be second only to the introduction of the electronic digital computer itself.

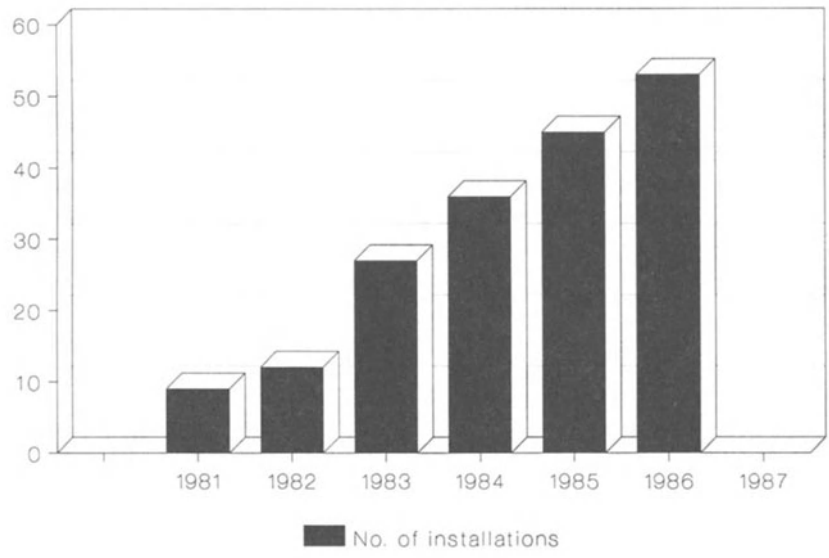


Figure 3. The growth of supercomputing installations in Europe (based on I. S. Duff, "Supercomputing in Europe-1987" in "Supercomputing", G. Goos and J. Hartmanis, eds., Lecture Notes in Computer Science **297**, 1031, 1987, Springer, Berlin).

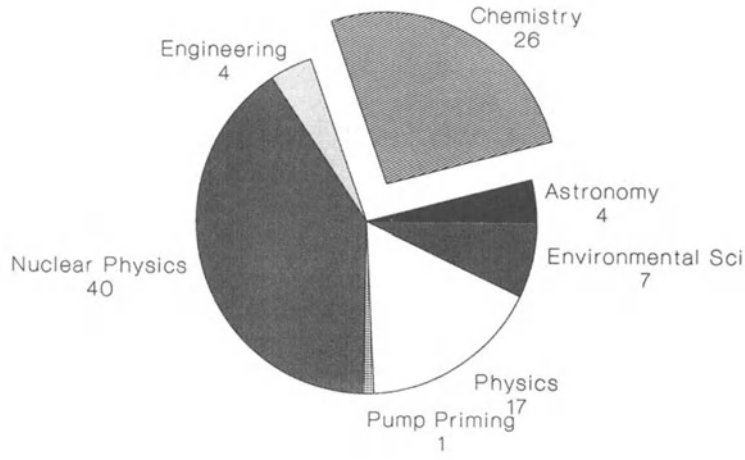


Figure 4. The use of supercomputers for chemical calculations. This figure shows the percentage of computer time devoted to chemical calculations on a typical supercomputer installation (based on statistics produced at the JRCSU at the Rutherford Appleton Laboratory).

That this is widely recognized, in general, is evident from the growth of so-called "supercomputing" installations over the last few years, which, as the data presented in Figure 3 show, has been exponential in Europe. In particular, chemical calculations account for a significant fraction of the processing time on these state-of-the-art machines, as the statistics presented in Figure 4 confirm.

This introductory chapter is intended to provide a background for the more detailed exposions given in the other chapters in this volume. In Section 2, we provide some background to the computational problems in chemistry which require the use of large-scale computing facilities. The theoretical limits to concurrent computation are discussed in Section 3. In Section 4, an overview of the wide range of computer architectures that are possible and/or available for concurrent computation is given, while, in Section 5, we address the difficult problem of algorithm design for concurrent computation. Finally, in Section 6, we look to the future.

2. Background

Most developed countries now recognize the importance of computing power to their scientific research programs. In the United Kingdom, the report[1] of a working party under the chairmanship of Professor A. J.

Forty on "Future Facilities for Advanced Research Computing," published in 1985, begins

> There are many problems that cannot readily be studied by experiment or are not amenable to a theoretical solution by mathematical analysis because of their complexity. It is possible, however, to approach a solution by numerical computation, given sufficient insight of the problem, a knowledge of the parameters involved and a sufficiently powerful computer.

The report goes on:

> The recognition of the potential of numerical computation as a research technique dates from the vision of von Neumann and others in the 1940s of the future use of digital computers as mathematical tools. However, the real emergence of computational science stems from the 1960s when powerful computers such as the Ferranti Atlas, a supercomputer of its day, became generally available.

Later, the authors write, "To provide a national resource and focus for advanced research computing one of the largest single computing installations in Great Britain was established ... the Atlas Computer Laboratory. ..." of the Rutherford Laboratory. "In its national role the Atlas computer was used to tackle some of the very large research problems of the day, for instance in quantum chemistry ... Crystallographers too were quick to exploit the machine's capabilities...". These and similar arguments have been used to establish powerful computational resources in many countries. Clearly, they are arguments that can find application in many areas of scientific endeavor.

Chemists have been among the leading participants in this research activity.[2] Table 1 serves to illustrate the use of state-of-the-art machines in chemical research. In it we list just some of the program packages that have been implemented on the Cray range of supercomputing machines. They range from *ab initio* quantum chemistry program suites capable of rather accurate calculations on relatively small molecules through semiempirical programs and macromolecular mechanics programs for the study of protein conformations, to programs that can simulate the properties of bulk systems.

The revolutionary changes in the practical applications of the methods of theoretical chemistry that occurred with the advent of the digital computer can be highlighted by the following recollections: One of the first tasks the author undertook as a postgraduate student in early 1971 was to implement on the IBM 360/195 at the Rutherford Laboratory a suite of programs written by Ruedenberg and his collaborators[3–6] for evaluating integrals over Slater functions to be used in electronic structure calculations. It was one of the fastest integral codes of its day. I recall describing this program to the late Professor C. A. Coulson and asking how long it

Table 1. Chemical Applications Software for CRAY Supercomputers[a]

AMBER: A macromolecular simulation package that includes a data base of amino acid and nucleic acid residues, a liquid water structure, and both united-atom and all-atom parameter sets

CASCADE: Calculates the structure and energy of a defect in an ionic crystal for a given potential model

CECTRP: Chemical equilibrium composition and transport properties program—calculates thermodynamic and transport properties of complex chemical mixtures

CHARMM: A macromolecular mechanics program that can minimize the energy, perform normal mode analysis or molecular dynamics simulations

GAUSSIAN: A quantum chemistry package with a broad range of uses

GRADSCF: A quantum chemistry package for predicting equilibrium structures, locating saddle points, calculating harmonic force constants and vibrational frequencies

LAZY PULVERIX: Simulates angle-dispersive x-ray and neutron powder diffraction pattern without the use of crystallographic tables

MADCAP: Analysis of distillation processes

MITHRIL: Solves crystal structures automatically from x-ray intensities by direct methods

MOLSCAT: Quantum mechanical calculation of elastic and inelastic molecular collision processes

MOPAC: A semiempirical molecular orbital package

XTAL: Programs for small molecule and macromolecular crystallographic calculations

[a] Based on the *Directory of Application Software for CRAY Supercomputers*, Cray Research, Inc.

had taken him to evaluate by hand computation the necessary integrals for his landmark molecular orbital calculation, published in 1938, on the ground state of the hydrogen molecule[7] in which he obtained the first accurate approximation to the Hartree–Fock orbitals. His answer was "One a day, and two a day when I got into the swing of things!" A similar point is made in the Foreword to the first volume of this series, where Professor R. McWeeny, one of Coulson's first students after the Second World War, relates how, at that time,[8] "Merely computing the integrals in readiness for a double zeta calculation on the ammonia molecule (today within the reach of a personal computer) would have taken about fifteen working years...." These recollections serve to underline the tremendous impact that the advent of the digital computer had on theoretical chemistry.

In 1952, Parr and Crawford organized a conference at Shelter Island[9] attended by Barnett, Boys, Coulson, Eyring, Hirschfelder, Kotani, Lennard-Jones, Lowdin, Mayer, Moffitt, Mulliken, Pitzer, Roothaan, Ruedenberg, Shull, Slater, and Van Vleck. At this meeting the need to develop accurate and reliable techniques for the evaluation of molecular integrals was widely recognized, and, what is more, the emphasis was on the use of electronic computers both in the integral evaluation and subsequent stages of electronic structure calculations. As Schaefer has

observed,[10] "the Shelter Island conference marked the emergence of theoretical chemistry from the 'dark ages' of the past decade or more."

Although the early computing machines allowed the computational chemists of the early 1950s to execute calculations that would have been impossible just a few years earlier, there was still a long way to go. The problem of evaluating integrals over exponential-type basis functions remained a problem, especially for polyatomic systems, and commanded the attention of some of the most gifted theoretical chemists for many years. Progress was steady rather than dramatic. From the period of the early 1950s until the late 1960s the main emphasis was on the calculation of energies and properties using orbital models and, in particular, in the self-consistent field approximation. From the late 1960s until the present time, the correlation problem has commanded a tremendous amount of attention. Theoretical chemistry provided the foundations for the new field of computational chemistry. Chemists added to their mathematical and theoretical tool kit expertise gathered from computer science and from numerical analysis. "Quantum chemistry," Lykos and Shavitt observed in 1981 in their Preface to *Supercomputers in Chemistry*,[8] which records the proceedings of the first symposium on the chemical applications of the new generation of machines to which this volume is devoted,

> has consumed vast quantities of computer time on machines of all sizes, and its potential for expanded computer resources is notorious. Furthermore, other fields of computational chemistry have been maturing, and their increasing demands for problem solving power have also been coming up against the constraints and limitations of current machines.

A year later, Lykos concludes that[12] "There is no forseeable limit to the complexity of the systems that chemists can model." And furthermore, "There is no forseeable scientific computer that chemistry modellers cannot exploit fully."

Recent years have seen a tremendous increase in the range and power of computers. State-of-the-art quantum chemical calculations have always required access to the most powerful computers of the day—supercomputers. According to the report on "Future Facilities for Advanced Research Computing,"[1]

> The term supercomputer is generally applied to machines which by virtue of their design yield better than expected performance for the technology of their day. They generally deliver this performance advantage over a limited but significant set of problems. Clearly, in the early 1960s the Ferranti Atlas would have qualified for this description. Subsequently the mantle passed to the CDC 7600 and (for floating point arithmetic) the IBM 360/195, which in turn have succumbed to machines like the Cray and Cyber 205 since the late 1970s.

Others reserve the term "supercomputer" for machines with distinctly

parallel architectures, of which the CRAY-1 can be said to be the first. Schafer records how[10]

> the world's first production supercomputer, the CRAY-1, was delivered to the Los Alamos Scientific Laboratory on April Fool's Day, 1976. For a considerable period of time, it was not obvious to theoretical chemists what was so "super" about this particular machine. More than two years after its appearance at Los Alamos, ... speeds for scalar operations were 2.3–2.8 times those achieved on the older Control Data Corporation 7600, while for vectorizable operations, factors of five could be achieved in some cases. ... the rewards were not appealing. That assessment changed radically a few years later after the arrival of the CRAY-1S (same machine, more memory) at the SERC Daresbury Laboratory.

It is in the nature of the electronic Schrödinger problem that new advances are to some extent "technology driven" if only in that theories are constructed with an eye to what is possible computationally. To a large extent technology determines what is feasible. Certain areas of research can be seen as arising entirely from the technical limitations of contemporary computing machines: Would there be any interest in developing techniques for making pseudopotential calculations if the corresponding all-electron calculations could be carried out straightforwardly? Would there be any point in devising new contraction schemes for basis sets of Gaussian-type functions if the primitive basis sets—or rather the two-electron integrals corresponding to these basis sets—could be handled without problem? McWeeny comments[8] "The first electronic computer I used was the "Whirlwind" at MIT in 1954; it solved my large ($20 \times 20!$) systems of secular equations in only a few minutes ..." Nearly 30 years later, Saunders and van Lenthe[13] described a "remarkably efficient CI method ... designed specifically for the CRAY-1S supercomputer" (quotation taken from Ref. 10), in which matrix eigenvalue problems of order 10^6 were solved for the lowest root.

At this point we should comment on the so-called minisupercomputers and superminicomputers. Although they can provide the computing power required to carry out a great deal of "standard" theoretical chemistry, there is a great danger that by relying exclusively on such machines we risk stagnation, that our horizons will be lowered, that theory will be limited by technology available, which will not be state-of-the-art technology. This has been the experience in the USA.[12] However, some of the more novel minisupercomputers contain large numbers of fairly low-power processors. These machines are valuable for exploring the computational science of computing in parallel.

It is, however, clear that, for the foreseeable future, "top-of-the-range" supercomputers will consist of a relatively small number of very powerful processors. Thus there are four processors in the CRAY X-MP/48, eight in the CRAY Y-MP, 16 in the CRAY 3, and 64 in the CRAY 4, projected for

1992. The ETA-10 machines, manufactured by ETA Systems/CDC, have up to eight processors at present, while the most powerful contemporary Japanese machines are uniprocessors. As the number of processors is increased, a rising fraction of them can become idle, either waiting for other processors to complete dependent computations or engaged in interprocessor communication.

We are not, however, solely concerned with CPU activity; Dongarra[14] claims that

> The key to using a high-performance computer effectively is to avoid unnecessary memory references. In most computers, data flows from memory into and out of registers and from registers into and out of functional units, which perform the given instructions on the data. Algorithm performance can be dominated by the amount of memory traffic rather than by the number of floating point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on data.

Is computational chemistry just a question of requiring more and more computing power? In a review of molecular quantum mechanics published in 1980, McWeeny and Pickup summarized the outlook as follows[15]:

> In looking to the future, one thing is clear beyond all doubt: ab initio molecular calculations, of "chemical accuracy," are going to be dominated more and more by the development of computers and highly efficient algorithms. New theories will still be required; ..., but formal theory will not be enough; the feasibility of the computational implementation will be of paramount importance.

But the present author emphasized the intricate connection between theory and computational practice, particularly when designing algorithms for concurrent computation[16]:

> ... a central problem of computational molecular physics is to develop efficient techniques for the ab initio treatment of electron correlation effects which allow the organization of concurrent computation on a very large scale. Clearly this ability, to organize an algorithm in such a way that calculation is performed in parallel, is not merely a computational problem but is most intricately connected with the theory upon which it is based.

For any given computational task we need to be able to determine the extent to which the various subtasks can proceed in parallel.

3. Concurrent Computation

In the first edition of their book *Parallel Computing*, published in 1981, Hockney and Jesshope observe that[17]

> What has changed with the advent of the parallel computer is the ratio between the performance of a good and bad computer program. This factor is not likely to exceed

a factor of two on a serial machine, whereas factors of ten or more are not uncommon on parallel machines.

Today the "factors of ten" can be replaced by factors of 100 or even more. Given the intricate connection between algorithms and the theory upon which they are based, it is clear that huge factors can emerge when the same quantity is calculated by different methods just because of the suitability of the methods to parallel computation. It is clearly of central importance to establish the extent to which parts of a computation can proceed concurrently, if at all.

It should be noted that there are some computations that cannot be performed more rapidly on N processors than on a single processor. A simple example is provided by the evaluation of[18]

$$x^{2^k}$$

by successive squaring; that is, by the recursion

$$y := x$$

$$y := y * y$$

Thus after one cycle we have x^2, after two cycles x^4, etc. and the required result can be obtained in k cycles on a single processor. Using N processors no power higher than

$$x^{2^m}$$

can be achieved in m cycles and thus the time required for the whole computation is independent of the number of processors available.

We have already likened the basic philosophy of concurrent computation to the old saying "Many hands make light work," but there is another saying that is equally relevant to the problem: "Too many cooks spoil the broth." In implementation on parallel machines it becomes vital to organize the use of not only the processing units but also the memory and the input–output channels. The performance of an algorithm can be dominated by the amount of memory traffic rather than the number of floating-point operations.

Following Dongarra *et al.*,[19] we find it useful to represent parallel computations graphically. A parallel program can be derived by breaking up the computation into discrete units and then constructing a *control flow graph.* The idea is probably best described by example. Let us suppose that we have nine subroutines, designated A, B, ..., I. A typical control flow graph is shown in Figure 5. In this figure, we show a scheme for subroutines A, B, ..., I in which we start by executing subroutines B, E, G, H, and I simultaneously since they appear as leaves in the control graph. Note that

there are three "copies" of subroutine B and four copies of subroutine I operating on different data. As soon as each of the routines I has been completed, the corresponding "copies" of F may execute. When execution of each of the F subroutines has been completed, execution of subroutine D can commence provided that both G and H have been completed. When D and E have completed execution, C may start. Finally, when the three "copies" of the subroutine B have been completed together with C, C may start. When A is completed the entire computation is finished.

A control graph for the evaluation of

$$x^{2^k}$$

is given in Figure 6. Here A represents the first step, that is, putting $y := x$, and B represents the successive squaring step $y := y * y$. Each execution of step B is dependent on the completion of the previous step. None of the steps in the computation can be executed concurrently. It is also clear from

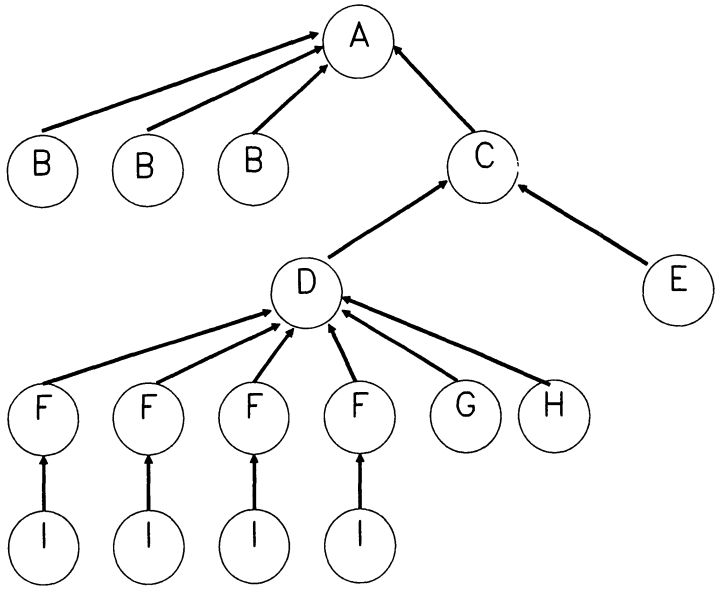


Figure 5. A control flow graph. Here we show a scheme for subroutines A, B, ..., I in which we start by executing subroutines B, E, G, H, and I simultaneously since they appear as leaves in the control graph. Note that there are three "copies" of subroutine B and four copies of subroutine I operating on different data. As soon each of the routines I has been completed, the corresponding "copies" of F may execute. When execution of each of the F subroutines has been completed, execution of subroutine D can commence provided that both G and H have been completed. When D and E have completed execution, C may start. Finally, when the three "copies" of the subroutine B have been completed together with C, C may start. When A is completed the entire computation is finished.

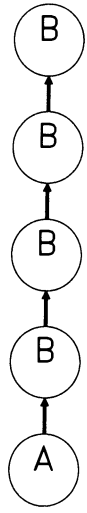


Figure 6. A control flow graph for the evaluation of x^{2^k}. A represents the first step, that is, putting $y := x$, and B represents the successive squaring step $y := y * y$. Each execution of step B is dependent on the completion of the previous step.

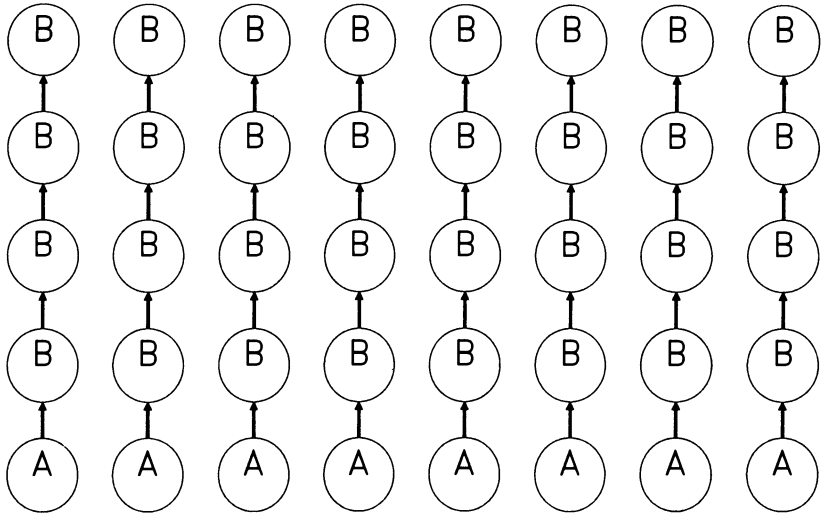


Figure 7. A control flow graph for the evaluation of $x_i^{2^k}$, $i = 1, 2, \ldots$.

Figure 6 that only one of the available processors is active at any one time. However, it is frequently the case in practical computations that we require the value of a function not for just one value of the argument x but for a number of values x_i, $i = 1, 2, \ldots$. A control graph for the evaluation of

$$x_i^{2^k}, \qquad i = 1, 2, \ldots$$

is displayed in Figure 7, from which the fact that most of the computation can proceed concurrently is immediately apparent. Thus, under the right circumstances, a computation that appears to be totally sequential can in fact upon closer examination be found to be amenable to concurrent computation. In other computations the possible degree of concurrency in the computation may be more immediately apparent.

4. Computer Architectures

4.1. Taxonomy

As noted in the Introduction to this chapter, the increase in speed of electronic computing machines in the period 1950–1970 was attributable to improvement in electronic engineering and to use of parallel computation. Until the early 1970s the parallel computation was to a large extent transparent to the computer user. Since that time, however, this has ceased to be the case and the user has found it necessary to familiarize himself with some of the details of the machine architecture in order to exploit the capabilities of the particular target machine effectively. It is essential to have a clear idea of the way in which the various processing units and the memory units are interconnected.

One of the first classification schemes for computers architectures is due to Flynn.[20] He divided machines as follows:

1. SISD: Single instruction stream—single data stream
2. SIMD: Single instruction stream—multiple data stream
3. MISD: Multiple instruction stream—single data stream
4. MIMD: Multiple instruction stream—multiple data stream

As explained, for example, by Hockney and Jesshope,[17] this is only one of a number of schemes, none of which gives a unique classification of architecture and none of which is, therefore, totally satisfactory. However, Flynn's taxonomy is widely used.

4.2. Serial Computers

The serial computer is the conventional von Neumann machine, which would be designated SISD in Flynn's classification scheme. There is one stream of instructions and each instruction lead to one operation thus leading to a stream of logically related arguments and results. This is the type of machine architecture that has dominated computational science for the past 40 years. It has had a significant influence on the design of the "traditional" algorithms employed in computational chemistry.

4.3. Vector Processors

The vector processing computer can be designated SIMD in Flynn's taxonomy. The CRAY 1 was the first commercially available machine of this type.

Consider, in detail, the task of adding two floating point numbers. For simplicity we assume that each number is represented by a mantissa and a decimal exponent. The addition can be divided into the following subtasks:

1. Determine the shift of one mantissa relative to the other in order to line up the decimal point.
2. Actually shift the mantissa to line up the decimal points.
3. Add the mantissa to obtain the mantissa of the result.
4. Normalize the result by shifting the mantissa so that the leading digit is next to the decimal point and the calculation of the exponent.

Obviously, these four subtasks must be executed sequentially since each subtask is dependent on the result of one or more previous subtasks. However, if we wish to add two vectors then the subtasks can be executed in parallel. As we illustrate in Figure 8, while subtask (1) is being carried out for the nth elements of the vectors, subtask (2) can be executed for the $(n-1)$th element, subtask (3) for the $(n-2)$th elements, and subtask (4) for the $(n-3)$th elements. n is then increased by 1 and the whole process repeated. This process is known as pipelining, and, after being initiated, it continues until all elements of the vectors have been processed or the number of operand registers associated with the functional unit have been exhausted. A pipeline works at its maximum efficiency if the number of operations to be processed is equal to an integer multiple of the number of operand registers. The "start up" of pipeline processing demands some extra time and the efficiency is, therefore, degraded for short vectors. Often it is necessary to rearrange randomly addressed data so that it is contiguous in memory. Vector processors currently afford the most powerful single processor machines.

4.4. Array Processors

In Flynn's taxonomy array processors may, like the vector processors described in Section 4.3, be described as SIMD machines.

An example of this type of architecture is provided by the ICL Distributed Array Processor (DAP). The DAP 510, which is now marketed by AMT, consists of 1024 single-bit processor elements arranged as a 32×32 array. Each processing element simultaneously executes the same instruction. Nearest neighbors are connected and a bus connects processors by rows and columns. Such machines have been used quite extensively in, for example, statistical mechanics, molecular simulation, and molecular graphics.

4.5. Parallel Processors

For the past decade or so, vector processors, such as the CRAY 1 and the CDC Cyber 205, have dominated the supercomputing scene. In more recent years, there has been a move towards multi-vector-processor shared

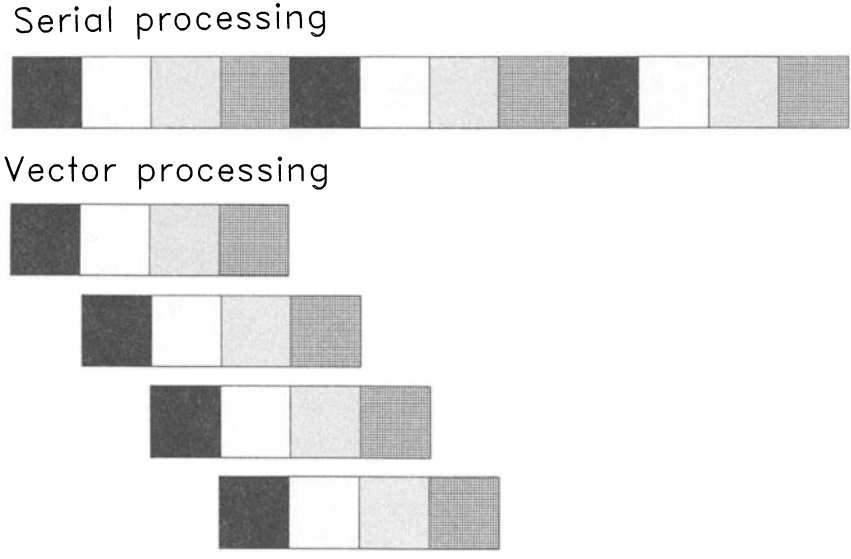


Figure 8. Addition of floating point numbers on a vector processor—a comparison of the four suboperations involved in the addition of two floating point numbers on a scalar processor and a vector processor. (1) Comparison of exponents (represented by a black square); (2) shifting of the mantissa (represented by a white square); (3) addition (represented by a lightly shaded square); (4) normalization of results (represented by a heavily shaded square).

memory machines. Examples of such machines include the CRAY X-MP, the more recent CRAY Y-MP, and the ETA-10. As we have already commented in Section 2, it is clear that, for the foreseeable future, "top-of-the-range" supercomputers will consist of a relatively small number of very powerful vector processors. It is also clear that, as scientific computations demand increasing computing power, the number of processors is set to increase and the need for the theoretical basis of a particular method and the associated computational algorithm to take account of such architectural changes will become increasingly important. Questions of interprocessor communications and connectivity will have to be addressed. It is essential, therefore, to have a clear idea of the way in which the various processing units and the memory units are interconnected. Hockney[21] has provided a useful structural taxonomy for parallel computers.

Parallel computers can, at the highest level, be basically divided into "networks" and "switched" systems.

A network system consists of a number of processors, each with its own local memory, which are connected together with an identifiable topology. Each processor and its local memory is referred to as a processing element. The processing elements can be connected together as a mesh, a cube, a hierarchical configuration, or in a reconfigurable array. The mesh

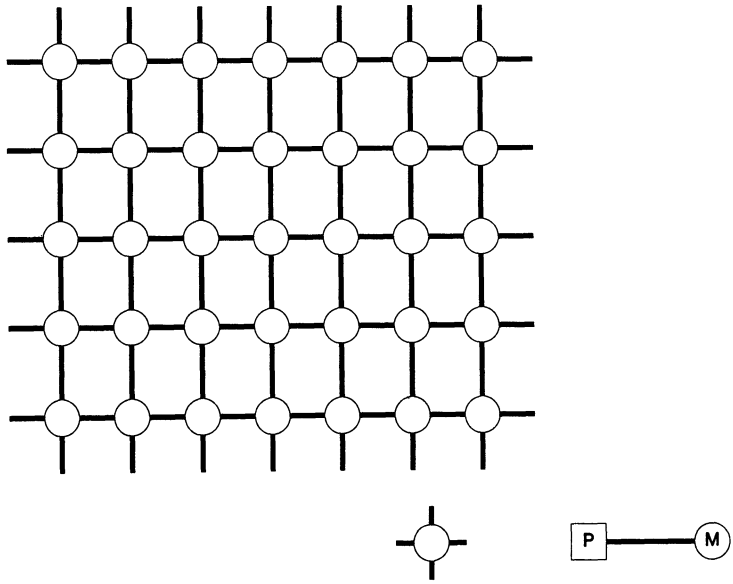


Figure 9. Mesh architecture. Each circle represent a processing element which consists of a processor, P, and associated memory, M.

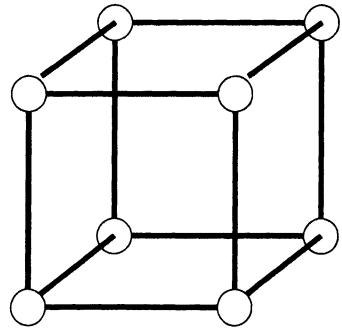


Figure 10. Cube architecture. Each circle represent a processing element which consists of a processor, P, and associated memory, M.

network, which is shown in Figure 9, contains some one-dimensional configurations, for example, rings, and some multidimensional configurations, for example, a square, hexagon, and other geometries. Cube networks, which are illustrated in Figure 10, range from pure hypercubes to "cube-connected-cycle" networks. The hypercube has particularly interesting and interconnection features. The n processor hypercube, the n-cube, consists of 2^n nodes, each associated with a single processing element and numbered by an n-bit binary number between 0 and 2^{n-1}. The processing elements are interconnected so that there is a link between two processors if and only if their binary representation differs by one and only one bit. For example, in the case $n = 3$, the eight processing elements can be represented as the vertices of a three-dimensional cube. The hypercube architecture

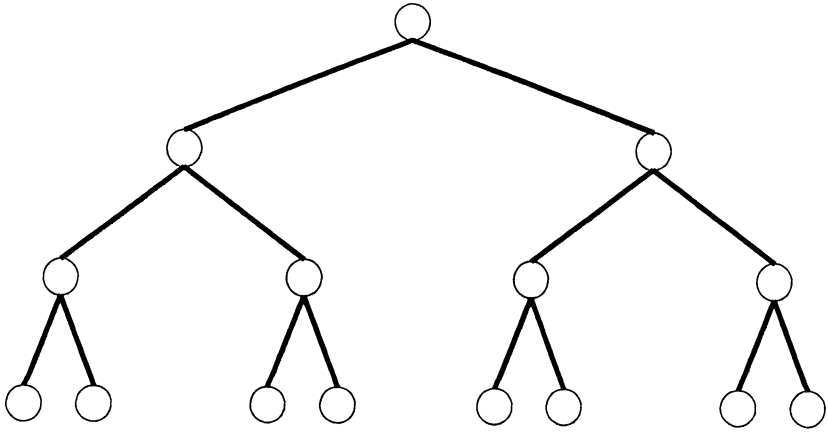


Figure 11. Hierarchical architecture. Each circle represent a processing element which consists of a processor, P, and associated memory, M.

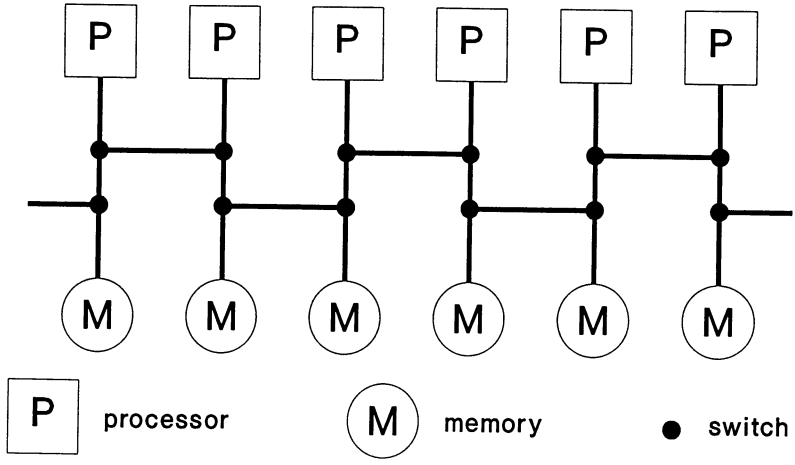


Figure 12. Shared memory architecture.

contains many classical topologies, such as two-dimensional meshes, in fact, it contains meshes of arbitrary dimension. It has homogeneity and symmetrical properties since no single processing element plays a particular role. In "cube-connected-cycle" networks each node of a cube is replaced by a ring or cycle of processing elements. Hierarchical networks are defined

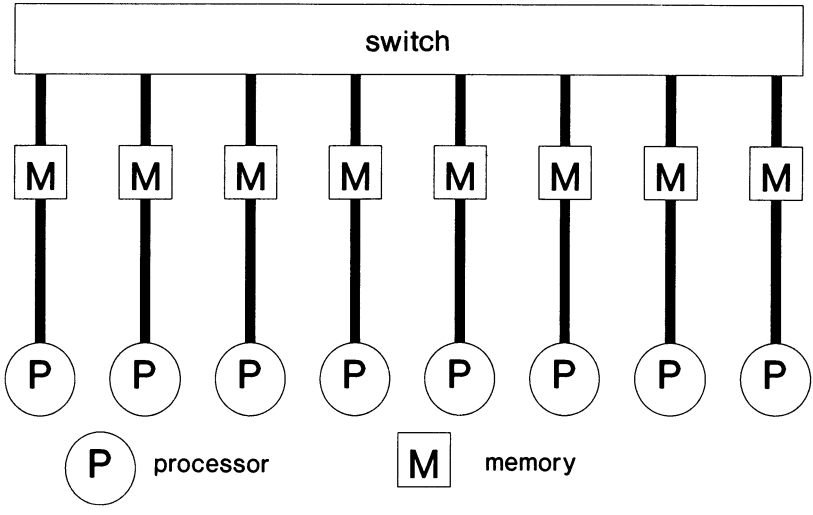


Figure 13. Distributed memory architecture.

in a recursive manner as we illustrate in Figure 11. This class of network architectures can be subdivided into tree networks, clusters of clusters, pyramids, etc. Finally, we have reconfigurable networks for which the interconnection pattern between the processing elements can be changed during program execution.

In a switched system there is a distinct unit that connects the processors and the memory units. The switch unit is responsible for all interconnections between the attached units. Switched systems can be subdivided into "shared memory" systems and "distributed memory" systems. In a shared memory system a number of processing units are connected via the switch unit to a number of independent memory units forming a common shared memory. This is illustrated in Figure 12. Each processing unit has its own local memory in a distributed memory system and the switch unit interconnects these processing units. This is illustrated in Figure 13.

5. Algorithms

5.1. The Paracomputer Model and Paracomputational Chemistry

5.1.1. The Paracomputer Model

In discussing the computational aspects of the electron correlation problem, the present author wrote[16]

> In order to adapt theories to take maximum advantage of parallel computers, it is perhaps preferable not to limit the consideration to one particular computer, but to consider the question of parallelism in more general terms. A very useful concept in such discussions is the paracomputer model.

A paracomputer[23] consists of a number of identical processors, each with a conventional order code set, that share a common memory which they can read simultaneously in a single cycle. Such machines cannot be physically realized, since no computing element can have more than some fixed number of external connections. The paracomputer model does, however, play a valuable theoretical role in devising algorithms for parallel computation and in the determination of the limits to parallel computation for a given problem.

As N-processor paracomputer can never be more than N times faster than a serial machine. The approach to this limit will be governed by the degree of parallelism in the algorithm and the theory upon which it is based. We have already noted that there are computations that can never be performed more rapidly on a N-processor paracomputer than on a

serial computer (recall the discussion given in Section 3). An important problem in computational chemistry is to investigate the extent to which existing theories lead to algorithms that are suited to parallel computation. The paracomputer model is, we submit, an extremely valuable concept in such considerations.

Although theoretically useful, the paracomputer model is not physically realizable since no computing element can have more than some fixed number of external connections. A second model, the ultracomputer,[23] is useful if one wishes to take account of this physical limitation. An ultracomputer consists of a number of processors, each with its own memory, but in which each processor can communicate with a fixed number, k, of other processors. Clearly, an N-processor ultracomputer cannot compute more rapidly than an N-processor paracomputer. The actual performance characteristics of the ultracomputer will depend on the communications characteristics of the problem and the processor interconnection scheme. No more than k quantities can be brought together during any cycle and, therefore, the solution of a problem involving N inputs and N outputs will require at least $O(\log_k N)$ cycles.

5.1.2. Paracomputational Chemistry

There is much advantage, in designing theories and algorithms for implementation on computing machines that are capable of carrying out parts of a calculation simultaneously, in considering the design of a method for the paracomputer model or, more realistically, the ultracomputer model. This defines what we shall term *paracomputational chemistry*—computational chemistry implemented on a paracomputer. In this way, the limits to parallel computation inherent in a particular approach can be readily identified. Machine specifics can be isolated.

5.2. Some Examples

In this section, we provide some practical illustrative example of the implementation of chemical algorithms on vector processing and parallel processing computers. Firstly, we consider the error function, the computational of which is required, for example, in electronic structure calculations employing basis sets of Gaussian-type functions. Secondly, we discuss matrix multiplication, which along with linear algebra in general, forms the central element of many computations. In the third and fourth sections we consider two aspects of the treatment of large molecules, an area of research that is opening up with the advent of increasingly powerful multiprocessing machines. In Section 5.3, we consider the problem of treating electron correlation in large molecules, a problem for which the linked

diagram theorem of many-body perturbation theory ensures that it is well suited. In Section 5.4 we consider large molecules of a different kind, those that involve heavier atoms and for which relativistic effects are, therefore, important.

5.2.1. *Example 1: The Error Function*

Gaussian functions are widely used in quantum chemical calculations to determine a finite basis set representation of molecular orbitals. Their popularity is attributable to the ease with which two-electron integrals involving such functions can be evaluated.[23] The bulk of the computation arising in the evaluation of these integrals is attributable to the evaluation of the function

$$F_{2n}(x) = x^{-(2n+1)/2} \int_0^{x^{1/2}} dt\, t^{2n} \exp(-t^2)$$

which is closely related to the error function. For example, the two-electron integral involving $1s$ Gaussian functions on centers **A**, **B**, **C**, and **D** with exponents α_A, α_B, α_C, and α_D, respectively, has the form

$$[s_A s_B | s_C s_D] = 2(\gamma/\pi)^{1/2} F_0[\gamma(\mathbf{P}-\mathbf{Q})^2] S_{AB} S_{CD}$$

where

$$\gamma = (\alpha_A + \alpha_B)(\alpha_C + \alpha_D)/(\alpha_A + \alpha_B + \alpha_C + \alpha_D)$$

$$\mathbf{P} = (\alpha_A \mathbf{R}_A + \alpha_B \mathbf{R}_B)/(\alpha_A + \alpha_B)$$

$$\mathbf{Q} = (\alpha_C \mathbf{R}_C + \alpha_D \mathbf{R}_D)/(\alpha_C + \alpha_D)$$

$$S_{AB} = (\alpha_A \alpha_B)^{-3/4} [2\alpha_A \alpha_B/(\alpha_A + \alpha_B)]^{3/2} \exp\{-[\alpha_A \alpha_B/(\alpha_A + \alpha_B)]\overline{AB}^2\}$$

$$S_{CD} = (\alpha_C \alpha_D)^{-3/4} [2\alpha_C \alpha_D/(\alpha_C + \alpha_D)]^{3/2} \exp\{-[\alpha_C \alpha_D/(\alpha_C + \alpha_D)]\overline{CD}^2\}$$

$$\overline{AB}^2 = (X_B - X_A)^2 + (Y_B - Y_A)^2 + (Z_B - Z_A)^2$$

$$\overline{CD}^2 = (X_D - X_C)^2 + (Y_D - Y_C)^2 + (Z_D - Z_C)^2$$

The error function is defined by

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x dt \exp(-t^2)$$

It can be accurately and efficiently evaluated using expansions in terms of

Tchebyshev polynomials.[24] Different expansions are employed for three different ranges of values of x. For $|x| \leqslant 2$ the expansion

$$\text{erf}(x) = x \sum_r{}' a_r T_r(t)$$

is employed where

$$t = \tfrac{1}{2}x^2 - 1$$

For $2 < |x| \leqslant x_h$ we put

$$\text{erf}(x) = \text{sign}(x)\left[1 - \frac{\exp(-x^2)}{|x|\sqrt{\pi}} \sum_r b_r T_r(t)\right]$$

where

$$t = \frac{(x-7)}{(x+3)}$$

Finally, for $|x| \geqslant x_h$ the expansion

$$\text{erf}(x) = \text{sign}(x)$$

is used where x_h is the value of x above which $|\text{erf}(x)| = 1$ within the rounding error of the machine being used.

In Figure 14 FORTRAN code suitable for evaluating the error function on a scalar processor is presented. The FUNCTION ERF(X) returns the value of the error function for the scalar argument X.

FORTRAN code suitable for implementation on a vector processor is shown in Figure 15. The FUNCTION VERF(X) returns the vector each element of which contains the error function for the corresponding element of the vector argument X. The conditional statements in the code given in Figure 14 inhibit efficient implementation on vector-processing computers. By replacing the scalar argument X by a vector argument much improved efficiency can be achieved. The approach is very similar to that taken in Section 3. The code shown in Figure 15 is written in FORTRAN 200, the dialect of FORTRAN that is employed on the Cyber 205 and ETA 10 machines. The use of the WHERE block in this code and the FORTRAN 200 vector notation should be noted. To illustrate the FORTRAN 200 vector syntax, which closely resembles the proposed FORTRAN 8X syntax,[25] consider the following simple DO loop:

```
      DO 1 I=1,N
      A(I+NA)=C+B(I+NB)
    1 CONTINUE
```

```
      REAL FUNCTION ERF(X)
C
C     Evaluation of the error function, erf(x)
C
C     .. Scalar Arguments ..
      REAL                  X
C     .. Local Scalars ..
      REAL                  CJ, CJP1, CJP2, HALF, ONE, SQRTPI, THREE,
     *                      TWENTY, TWO, X2, XUP, XV, ZERO
      INTEGER               J, NA, NB
C     .. Local Arrays ..
      REAL                  A(15), B(15)
C     .. Intrinsic Functions ..
      INTRINSIC             ABS, EXP, SIGN
C     .. Data statements ..
      DATA NA,NB/15,15/,XUP/5.75E0/,SQRTPI/1.7724538509055/
     A,A(1),A(2),A(3),A(4),A(5),A(6),A(7),A(8),A(9),A(10)
     A,A(11),A(12),A(13),A(14),A(15)
     A/1.9449071068179,4.20186582324E-2,-1.86866103977E-2
     A,5.1281061839E-3,-1.0683107462E-3,1.744737872E-4
     A,-2.15642056E-5,1.7282658E-6,-2.00479E-8,-1.64782E-8
     A,2.0008E-9,2.58E-11,-3.06E-11,1.9E-12,4.0E-13/
     A,B(1),B(2),B(3),B(4),B(5),B(6),B(7),B(8),B(9),B(10)
     A,B(11),B(12),B(13),B(14),B(15)
     A/1.4831105640848,-3.010710733866E-1,6.899483068988E-2
     A,-1.391627126472E-2,2.4207995224E-3,-3.658639686E-4
     A,4.86209844E-5,-5.7492565E-6,6.113243E-7,-5.89910E-8
     A,5.2070E-9,-4.233E-10,3.19E-11,-2.2E-12,1.0E-13/
      DATA ZERO, ONE, TWO, THREE, TWENTY, HALF /0.0,1.0,2.0,3.0,20.0,
     * 0.5/
C     .. Executable Statements ..
      XV = ABS(X)
      IF (XV.GE.XUP) GO TO 600
      IF (XV.LE.TWO) GO TO 300
      X2 = TWO - TWENTY/(XV+THREE)
C
      CJP2 = ZERO
      CJP1 = A(NA)
      J = NA - 1
  100 CONTINUE
      CJ = X2*CJP1 - CJP2 + A(J)
      IF (J.EQ.1) GO TO 40
      CJP2 = CJP1
      CJP1 = CJ
      J = J - 1
      GO TO 100
  200 CONTINUE
      X2 = HALF*(CJ-CJP2)/XV*EXP(-X*X)/SQRTPI
      ERF = (ONE-X2)*SIGN(ONE,X)
      GO TO 700
```

Figure 14. FORTRAN code suitable for the evaluation of the error function on a scalar processor.

```
C
  300 CONTINUE
      X2 = X*X - TWO
      CJP2 = ZERO
      CJP1 = B(NB)
      J = NB - 1
  400 CONTINUE
      CJ = X2*CJP1 - CJP2 + B(J)
      IF (J.EQ.1) GO TO 500
      CJP2 = CJP1
      CJP1 = CJ
      J = J - 1
      GO TO 400
  500 CONTINUE
      ERF = HALF*(CJ-CJP2)*X
      GO TO 700
C
  600 CONTINUE
      ERF = SIGN(ONE,X)
  700 RETURN
      END
```

Figure 14. *(Continued)*

In FORTRAN 200 vector syntax this becomes

```
A(NA;N) = C + B(NB;N)
```

where NA and NB are the starting addresses for A and B, respectively, and the length of the vector is given after the semicolon. Intrinsic functions such as ABS are available in vector form. For example, the statement

```
Y(1;N) = VABS(X(1;N);Y(1;N))
```

results in each element of the vector Y(1;N) containing the absolute value of the corresponding element of the vector X(1;N). The WHERE block is a particularly powerful feature of FORTRAN 200. In its simplest form it has the form

```
WHERE (LBIT(1;N))
A(1:N) = B(1;N)
ENDWHERE
```

in which LBIT(1;N) is a BIT vector whose elements are either .TRUE. or corresponding elements of A(1;N) according to the elements of LBIT(1;N). In Figure 16, the code that is equivalent to that given in Figure 15 is

```
      REAL FUNCTION VERF(X;*)
C
C     Evaluation of the error function, erf(x)
C
C     FORTRAN 200 version for CYBER 205 and ETA 10 machines
C
C     .. Array arguments ..
      REAL X(100)
C     .. Local Scalars ..
      REAL                  HALF, ONE, SQRTPI, THREE,
     *                      TWENTY, TWO, XUP, ZERO
      INTEGER               J, NA, NB
C     .. Local Arrays ..
      REAL X2(100),XV(100),CJ(100),CJP1(100),CJP2(100),V(100),
     *                      A(15), B(15)
C     .. Intrinsic Functions ..
      INTRINSIC             ABS, EXP, SIGN
      DESCRIPTOR VERF
C     .. Data statements ..
      DATA NA,NB/15,15/,XUP/5.75E0/,SQRTPI/1.7724538509055/
     A,A(1),A(2),A(3),A(4),A(5),A(6),A(7),A(8),A(9),A(10)
     A,A(11),A(12),A(13),A(14),A(15)
     A/1.9449071068179,4.20186582324E-2,-1.86866103977E-2
     A,5.1281061839E-3,-1.0683107462E-3,1.744737872E-4
     A,-2.15642056E-5,1.7282658E-6,-2.00479E-8,-1.64782E-8
     A,2.0008E-9,2.58E-11,-3.06E-11,1.9E-12,4.0E-13/
     A,B(1),B(2),B(3),B(4),B(5),B(6),B(7),B(8),B(9),B(10)
     A,B(11),B(12),B(13),B(14),B(15)
     A/1.4831105640848,-3.010710733866E-1,6.899483068898E-2
     A,-1.39162712647E-2,2.4207995224E-3,-3.658639686E-4
     A,4.86209844E-5,-5.7492565E-6,6.113243E-7,-5.89910E-8
     A,5.2070E-9,-4.233E-10,3.19E-11,-2.2E-12,1.0E-13/
C
C
      DATA ZERO, ONE, TWO, THREE, TWENTY, HALF /0.0,1.0,2.0,3.0,20.0,
     * 0.5/
C     .. Executable Statements ..
      N=100
C
      CJ(1:N)=1.0
      XV(1;N)=VABS(X(1;N);XV(1;N))
      V(1;N)=VSIGN(CJ(1;N),X(1;N);V(1;N))
C
      X2(1;N)=X(1;N)*X(1;N)-2.0
      CJP2(1:N)=0.0
      CJP1(1:N)=B(NB)
      DO 100, J=NB-1,2,-1
      CJ(1;N)=X2(1;N)*CJP1(1;N)-CJP2(1;N)+B(J)
      CJP2(1;N)=CJP1(1;N)
      CJP1(1;N)=CJ(1;N)
```

Figure 15. FORTRAN 200 code suitable for the evaluation of the error function on the Cyber 205 and ETA-10 computers using the vector processing facility.

```
100 CONTINUE
    CJ(1;N)=X2(1;N)*CJP1(1;N)-CJP2(1;N)+B(1)
    WHERE (XV(1;N).LE.2.0)
    V(1;N)=HALF*(CJ(1;N)-CJP2(1;N))*X(1;N)
    ENDWHERE
    X2(1;N)=2.0-20.0/(XV(1;N)+3.0)
    CJP2(1:N)=0.0
    CJP1(1:N)=A(NA)
    DO 200, J=NA-1,2,-1
    CJ(1;N)=X2(1;N)*CJP1(1;N)-CJP2(1;N)+A(J)
    CJP2(1;N)=CJP1(1;N)
    CJP1(1;N)=CJ(1;N)
200 CONTINUE
    CJ(1;N)=X2(1;N)*CJP1(1;N)-CJP2(1;N)+A(1)
    X2(1;N)=-X(1;N)*X(1;N)
    X2(1;N)=VEXP(X2(1;N);X2(1;N))
    X2(1;N)=0.5*(CJ(1;N)-CJP2(1;N))/XV(1;N)*X2(1;N)/SQRTPI
    CJP1(1;N)=1.0
    CJ(1;N)=VSIGN(CJP1(1;N),X(1;N);CJ(1;N))
    WHERE (.NOT.((XV(1;N).GE.XUP).OR.(XV(1;N).LE.TWO)))
    V(1;N)=(CJP1(1;N)-X2(1;N))*CJ(1;N)
    ENDWHERE
    VERF=V(1;N)
    RETURN
    END
```

Figure 15. (*Continued*)

presented using the DESCRIPTOR notation of FORTRAN 200. In this notation, the above simple WHERE block would take the form

```
WHERE (LBIT)
A = B
ENDWHERE
```

LBIT, A, and B are declared DESCRIPTORS. The code given in Figure 16 has an advantage over that given in Figure 15 in that the work arrays that are required are assigned dynamically. The intrinsic function Q8SLEN returns the length of the argument vector X. The ASSIGN statements then allocate memory to the various work arrays. The DESCRIPTOR syntax is used throughout and thus X(1;N) is replaced by X where X is declared a DESCRIPTOR.

In Table 2, a comparison is made of the time required on the Cyber 205 vector processor (two pipelines) using the "scalar" code given in Figure 14 and the code given in Figure 16. These timing tests were performed on the Cyber 205 installation at the University of Manchester Regional Computer Centre.

```
      REAL FUNCTION VERF(X;*)
C
C     Evaluation of the error function, erf(x)
C
C     FORTRAN 200 version for the CYBER 205 and ETA 10 using descriptors
C     .. Array arguments ..
C     REAL X(100)
C     .. Local Scalars ..
      REAL                HALF, ONE, SQRTPI, THREE,
     *                    TWENTY, TWO, XUP, ZERO
      INTEGER             J, NA, NB
C     .. Local Arrays ..
C     REAL X2(100),XV(100),CJ(100),CJP1(100),CJP2(100),V(100)
     *                    A(15), B(15)
C     .. Descriptors ..
      DESCRIPTOR VERF,X,XV,X2,CJ,CJP1,CJP2,V
C     .. Data statements ..
C
      DATA NA,NB/15,15/,XUP/5.75E0/,SQRTPI/1.7724538509055/
     A,A(1),A(2),A(3),A(4),A(5),A(6),A(7),A(8),A(9),A(10)
     A,A(11),A(12),A(13),A(14),A(15)
     A/1.9449071068179,4.20186582324E-2,-1.86866103977E-2
     A,5.1281061839E-3,-1.0683107462E-3,1.744737872E-4
     A,-2.15642056E-5,1.7282658E-6,-2.00479E-8,-1.64782E-8
     A,2.0008E-9,2.58E-11,-3.06E-11,1.9E-12,4.0E-13/
     A,B(1),B(2),B(3),B(4),B(5),B(6),B(7),B(8),B(9),B(10)
     A,B(11),B(12),B(13),B(14),B(15)
     A/1.4831105640848,-3.010710733866E-1,6.89948306898E-2
     A,-1.39162712647E-2,2.4207995224E-3,-3.658639686E-4
     A,4.86209844E-5,-5.7492565E-6,6.113243E-7,-5.89910E-8
     A,5.2070E-9,-4.233E-10,3.19E-11,-2.2E-12,1.0E-13/
C
C
C     .. Executable Statements ..
      IFAIL=0
      N=Q8SLEN(X)
      ASSIGN XV,.DYN.N
      ASSIGN X2,.DYN.N
      ASSIGN CJ,.DYN.N
      ASSIGN CJP1,.DYN.N
      ASSIGN CJP2,.DYN.N
      ASSIGN V,.DYN.N
C
      XV=VABS(X;XV)
C
      X2=X*X-2.0
      CJP1=X2*B(NB)+B(NB-1)
      CJP2=B(NB)
      DO 100, J=NB-2,2,-1
      CJ=X2*CJP1-CJP2+B(J)
      CJP2=CJP1
      CJP1=CJ
```

Figure 16. Alternative FORTRAN 200 code, uses the DESCRIPTOR language element, suitable for the evaluation of the error function on the Cyber 205 and ETA-10 computers using the vector processing facility.

```
100 CONTINUE
    CJ=X2*CJP1-CJP2+B(1)
    WHERE (XV.LE.2.0)
    V=0.5*(CJ-CJP2)*X
    ENDWHERE

    X2=2.0-20.0/(XV+3.0)
    CJP1=X2*A(NA)+A(NA-1)
    CJP2=A(NA)
    DO 200, J=NA-2,2,-1
    CJ=X2*CJP1-CJP2+A(J)
    CJP2=CJP1
    CJP1=CJ
200 CONTINUE
    CJ=X2*CJP1-CJP2+A(1)
    X2=-X*X
    X2=VEXP(X2;X2)
    X2=0.5*(CJ-CJP2)/XV*X2/SQRTPI
    CJP1=1.0-X2
    CJ=VSIGN(CJP1,X;CJ)
    WHERE (XV.GT.2.0)
    V=CJ
    ENDWHERE
    VERF=V
    FREE
    RETURN
    END
```

Figure 16. *(Continued)*

It should be underlined that, in practical evaluation of molecular integrals over Gaussian-type functions, the $F_{2n}(x)$ are computed directly rather than from the error function.[23] However, the procedure is similar to that described above in that different prescriptions are employed for different ranges of the arguments.

5.2.2. Example 2: Matrix Multiplication

Many chemical calculations involve extensive linear algebra. Such calculations can be efficiently executed on modern supercomputers by exploiting a set of basic linear algebra subroutines (BLAS) (see, for example, Ref. 26–31) tailored to the particular target machine. These subroutines are sometimes divided into three types:

1. Level 1 BLAS—vector operations;
2. Level 2 BLAS—matrix-vector operations;
3. Level 3 BLAS—matrix operations.

In this section, we consider as an example one of the most fundamental operations of linear algebra matrix multiplication. Consider the product of

Table 2. Central Processing Time Required to Evaluate the Error Function, erf(x_i), $i = 1,...,N$ on a Cyber 205 (Two Pipeline) Computers Using Scalar Instructions and Vector Instructions

N	Scalar code time (ms)	Vector code time (ms)
500	9.4	1.6
1000	18.6	2.4
1500	27.8	3.4
2000	37.7	4.5
2500	46.4	5.6
3000	55.4	6.6
3500	64.1	7.7
4000	73.2	8.8
4500	82.8	9.9
5000	92.5	11.0
6000	109.5	13.6
7000	129.2	16.0
8000	147.7	18.7
9000	166.1	21.2
10000	183.3	23.5

two square matrices $\mathbf{A} = \mathbf{BC}$, which can be written in terms of the matrix elements as

$$A_{ij} = \sum_k B_{ik} C_{kj}$$

FORTRAN code for evaluating this product can be written in at least three distinct ways. In each of the following cases we assume that the elements of the array A have been set to zero. First, we have what might be termed the inner product method, which is clearly sequential, since it has the form of a scalar summation:

```
*
*    Matrix multiplication using a scalar loop
*    structure.
*
     DO 1 I=1,N
      DO 2 J=1,N
       DO 3 K=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
3     CONTINUE
2     CONTINUE
1    CONTINUE
```

Next we have the middle product method, which has a loop structure suitable for vector processing computers:

```
*
*    Matrix multiplication using a loop
*    structure suitable for vector processors.
*
     DO 1 J=1,N
      DO 2 K=1,N
       DO 3 I=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
3    CONTINUE
2    CONTINUE
1    CONTINUE
```

The inner loop in this construction can be executed in vector mode. Finally, we have the outer product method, which has the following form:

```
*
*    Matrix multiplication using a loop
*    structure suitable for array processors.
*
     DO 1 K=1,N
      DO 2 I=1,N
       DO 3 J=1,N
        A(I,J)=A(I,J)+B(I,K)*C(K,J)
3    CONTINUE
2    CONTINUE
1    CONTINUE
```

Whereas the middle product method can be said to have a degree of parallelism n, the outer product method can be seen to have a degree of parallelism n^2, since the inner two loops are independent. In Table 3 we present the results of timing tests for the inner product, middle product, and outer product matrix multiplication schemes performed on a single processor of the CRAY X-MP/48. Both the middle product scheme and the outer product scheme are able to exploit the vector processing capabilities of this machine, and this is reflected in the times quoted. A corresponding set of results for a single processor ETA-10G implementation is given in Table 4.

Figure 17 illustrates the partitioning of the matrices **A**, **B**, and **C** for the multitasking of a single matrix multiplication. Given matrices **A** and **B** of dimension $n_1 \times m$ and $m \times n_2$, respectively, we wish to compute the matrix product $\mathbf{C} = \mathbf{AB}$ in which **C** is of dimension $n_1 \times n_2$. The partitions

of the matrices **A**, **B**, and **C** shown in Figure 17 are of dimension $p_1 \times p_2$, $p_2 \times n_2$, and $p_1 \times n_2$, respectively. One block row of the matrix **C** is updated according to

$$C_{i,*} \leftarrow C_{i,*} + A_{i,j} B_{j,*}$$

FORTRAN code for the simultaneous multiplication of several distinct matrices on a multitasking computer, again the CRAY X-MP/48, is given in Figure 18. This code is applicable to an arbitrary number of processors, although in the present discussion we are, of course, limited to four physical processors and we shall assume this limit for both the number of physical processors and the number of logical processors in the immediate

Table 3. Results of Timing Tests for the Inner Product, Middle Product, and Outer Product Matrix Multiplication Schemes Performed on a Single Processor of the CRAY X-MP/48[a]

	CPU time (s)		
Dimension, N	Inner product	Middle product	Outer product
20	0.001	0.000	0.000
40	0.007	0.002	0.002
60	0.019	0.005	0.005
80	0.036	0.012	0.011
100	0.060	0.021	0.021
120	0.093	0.034	0.033
140	0.139	0.054	0.055
160	0.187	0.075	0.074
180	0.245	0.106	0.105
200	0.324	0.146	0.144
220	0.397	0.189	0.190
240	0.486	0.241	0.240
260	0.622	0.316	0.310
280	0.737	0.371	0.380
300	0.875	0.454	0.458
320	1.039	0.540	0.550
340	1.191	0.666	0.659
360	1.387	0.774	0.772
380	1.591	0.899	0.898
400	1.852	1.059	1.060
420	2.095	1.214	1.216
440	2.367	1.376	1.349
460	2.661	1.579	1.587
480	3.013	1.817	1.835
500	3.321	1.946	1.964

[a] The matrices are taken to be square and have dimension N.

Table 3. (*Continued*)

The inner product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 101 I=1,N
      DO 102 J=1,N
      DO 103 K=1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
103   CONTINUE
102   CONTINUE
101   CONTINUE
      T3=SECOND( )
      TI=T3-T2-(T2-T1)
```

The middle product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 201 J=1,N
      DO 202 K=1,N
      DO 203 I=1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
203   CONTINUE
202   CONTINUE
201   CONTINUE
      T3=SECOND( )
      TM=T3-T2-(T2-T1)
```

The outer product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 301 K=1,N
      DO 302 J=1,N
      DO 303 I=1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
303   CONTINUE
302   CONTINUE
301   CONTINUE
      T3=SECOND( )
      TO=T3-T2-(T2-T1)
```

discussion of the code. The code in Figure 18 employs Cray macrotasking facilities to calculate each of the matrix products as a separate task. Clearly, we only wish to initialize four tasks at any one time since otherwise the tasks will be competing for resources. TASK(*, *) is the task control array. The vectors TASK(1, n), TASK(2, n), TASK(3, n),

$n = 1, 2, 3, 4$, correspond to each of the tasks being performed at a given time. TSKTUNE modifies tuning parameters in the library scheduler and is called during initialization to establish the number of processors to be used. Each call to TSKSTART initializes a task. The arguments of TSKSTART are the task control array, the external entry, and a list of arguments. TSKWAIT monitors the completion of the tasks. It should be noted that the Cray macrotasking feature allows control over the number of logical processors and not the number of physical processors. Timing data for the code presented in Figure 18 are shown in Figure 19.

Table 4. Results of Timing Tests for the Inner Product, Middle Product and Outer Product Matrix Multiplication Schemes Performed on a Single Processor of the ETA-10G[a]

	CPU time (s)		
Dimension, N	Inner product	Middle product	Outer product
20	0.001	0.000	0.000
40	0.004	0.001	0.001
60	0.010	0.003	0.003
80	0.021	0.005	0.005
100	0.036	0.009	0.009
120	0.057	0.014	0.014
140	0.190	0.021	0.021
160	0.285	0.030	0.029
180	0.398	0.040	0.039
200	0.522	0.052	0.052
220	0.687	0.067	0.066
240	0.921	0.082	0.082
260	1.134	0.099	0.103
280	1.405	0.121	0.125
300	1.726	0.144	0.148
320	2.172	0.172	0.177
340	2.550	0.203	0.209
360	3.014	0.234	0.241
380	3.541	0.272	0.280
400	4.054	0.314	0.323
420	4.887	0.357	0.367
440	5.620	0.404	0.416
460	6.417	0.461	0.474
480	7.184	0.516	0.530
500	8.132	0.575	0.591

[a] The matrices are taken to be square and have dimension N.

Table 4. (*Continued*)

The inner product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 101 I=1,N
      DO 102 J=1,N
      A(I,J)=A(I,J)+Q8SDOT(Q8VGATHP(B(I,1;N),700,N;N),C(1,J;N))
  102 CONTINUE
  101 CONTINUE
      T3=SECOND( )
      TI=T3-T2-(T2-T1)
```

The middle product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 201 J=1,N
      DO 202 K=1,N
      A(1,J;N)=A(1,J;N)+B(1,K;N)*C(K,J)
  202 CONTINUE
  201 CONTINUE
      T3=SECOND( )
      TM=T3-T2-(T2-T1)
```

The outer product scheme was implemented with the following code:

```
      T1=SECOND( )
      T2=SECOND( )
      DO 301 K=1,N
      DO 302 J=1,N
      A(1,J;N)=A(1,J;N)+B(1,K;N)*C(K,J)
  302 CONTINUE
  301 CONTINUE
      T3=SECOND( )
      TO=T3-T2-(T2-T1)
```

A compiler option preventing reordering of the DO loops was invoked.

5.2.3. *Example 3: Many-Body Perturbation Theory for Extended Systems*

Over the past 20 years, the diagrammatic many-body perturbation theory has emerged as one of the principal methods for describing electron correlation in atomic and molecular systems, its advantages over the traditional configuration interaction method being widely recognized.[32–36] From a theoretical point of view, the linked diagram theorem of many-

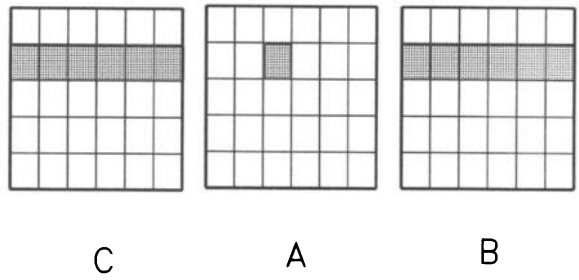


Figure 17. Partition of matrices for matrix multiplication on multiprocessing computers. (See text for details.)

body perturbation theory ensures that calculated energies scale linearly with the number of electrons in the system, whereas, from a computational point of view, it not only affords a noniterative algorithm for the calculation of accurate electron correlation energies but also leads to algorithms that are particularly well suited to concurrent computation. Recent surveys of atomic many-body perturbation theory studies have been given by Jankowski,[32] Lindgren and Morrison,[33] and Wilson.[34] Molecular applications have been reviewed by Urban *et al.*[35] and by Wilson.[36] Second-order calculations can often afford a very accurate description of electron correlation effects in systems for which a single determinant reference function is appropriate.[37]

In nonrelativistic many-body perturbation theory there are two sources of error: that arising from truncation of the perturbation series and that associated with basis set truncation. The situation is illustrated schematically in Figure 20. A fourth-order diagram in the many-body perturbation theory series for the correlation energy of a closed shell system involving a triply excited intermediate state is shown in Figure 21. The calculation of the energy components corresponding to these diagrams is the most computationally demanding part of fourth-order many-body perturbation theory studies. A decade ago, it was demonstrated that these energy components could be determined efficiently on vector processing machines.[38–40] However, it is well established, particularly for polyatomic molecular systems, that basis set truncation errors are frequently far more important than the neglect of higher-order terms in the perturbation expansion,[41] and it is suggested that the algorithms proposed in this section, which are based on the recent work of Moncreiff *et al.*,[37] will lead to the reduction of these errors by allowing larger and more flexible basis sets to be considered in second-order calculations than in current state-of-the-art calculations.

```
      PROGRAM MULTIMAT
      PARAMETER(N=256, NCPUS= 4, MTSK=100, NTSK=8)
      INTEGER TASK
      DIMENSION DUMP(200)
      COMMON /A/ A(N,N),B(N,N),C(N,N)
      COMMON /TSK / TASK (3,NCPUS)
      EXTERNAL MXMA
      N1=1
      NN=N
      DO 1 I=1,NCPUS-1
      TASK(1,I)=3
      TASK(3,I)=I
    1 CONTINUE
      IF (NCPUS.GT.1) THEN
      CALL TSKTUNE('MAXCPU',NCPUS)
      CALL TSKTUNE('DBRELEAS',NCPUS)
      ENDIF
C
      DO 2 I=1,N
      DO 2 J=1,N
    2 A(I,J)=0.987654321
C
      I=0
      CALL PERF('RESET'L,0,DUMP,200)
      CALL PERF('ON'L,0,DUMP,200)
      DO 3 K=1,MTSK
      I=I+1
      IF(I.NE.NTSK) THEN
          CALL TSKSTART(TASK(1,I),MXMA ,
     1    A(1,1),N1,NN,B(1,1),N1,NN,C(1,1),N1,NN,NN,NN,NN)
      ELSE
          CALL MXMA (
     1    A(1,1),N1,NN,B(1,1),N1,NN,C(1,1),N1,NN,NN,NN,NN)
          DO  4 J=1,NCPUS-1
    4     CALL TSKWAIT(TASK(1,J))
          I=0
      ENDIF
    3 CONTINUE
          DO  6 J=1,I
    6     CALL TSKWAIT(TASK(1,J))
      CALL PERF('OFF'L,0,DUMP,200)
      CALL PERF('RESET'L,0,DUMP,200)
      CALL PERFPRT(DUMP,200,'FT06'L)
      STOP
      END
```

Figure 18. Matrix multiplication code for the CRAY X-MP/48.

The second-order energy for a closed shell system may be written

$$E_2 = \tfrac{1}{4} \sum_{ij} \sum_{ab} \langle ij|\, 0\, |ab\rangle \langle ab|\, 0\, |ij\rangle / (\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b)$$

where the two-electron integrals in the numerator are defined by

$$\langle pq|\, 0\, |rs\rangle = \int d\tau_1 \int d\tau_2\, \phi_p^*(\tau_1)\, \phi_q^*(\tau_2)\, r_{12}^{-1} [\phi_r(\tau_1)\, \phi_s(\tau_2) - \phi_r(\tau_2)\, \phi_s(\tau_1)]$$

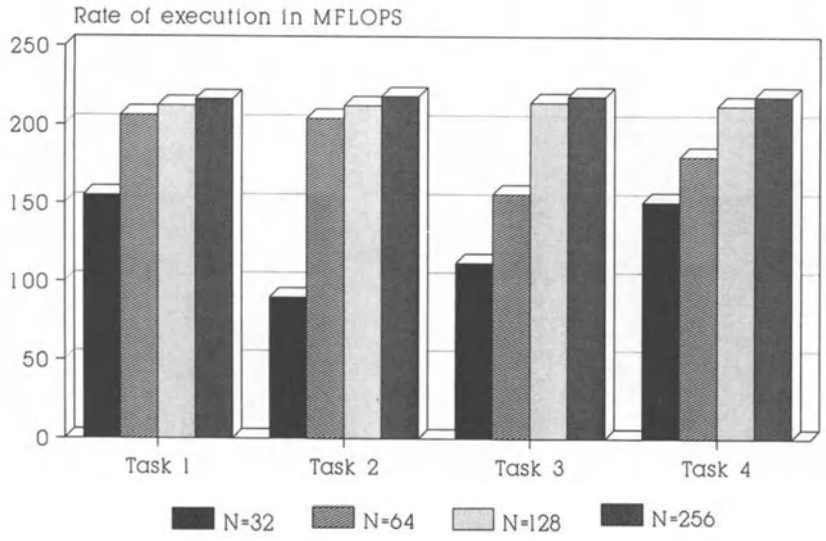


Figure 19. Rate of execution for matrix multiplication on the CRAY X-MP/48. Four tasks, that is, matrix multiplications, are carried out concurrently on the four processors of the CRAY X-MP/48. The rate of execution is given for matrices of dimension, $N = 32, 64, 128, 256$.

in which ϕ_p is a spin-orbital and τ_p denotes the coordinates, space and spin, of the pth electron, and the ε_p are orbital energies. In this work we use lower case indices p, q, r, ... to denote arbitrary spin-orbitals; the indices i, j, k, ... are employed for occupied spin-orbitals; and a, b, c, ... are used for unoccupied spin-orbitals.

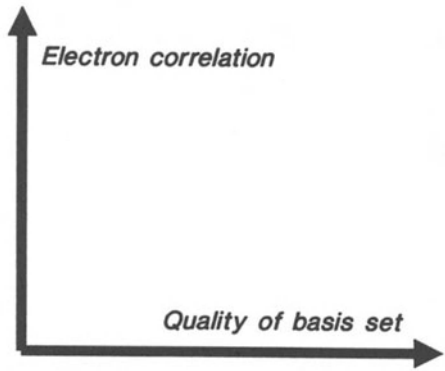


Figure 20. Truncation of the electron correlation energy expansion and the basis set in atomic and molecular electronic calculations. Ideally both the quality of the basis set and the description of electron correlation should be improved represented by the top right-hand corner in the above figure. In practice, computational restrictions mean that if the basis set is improved then the description of correlation effects is restricted and vice versa.

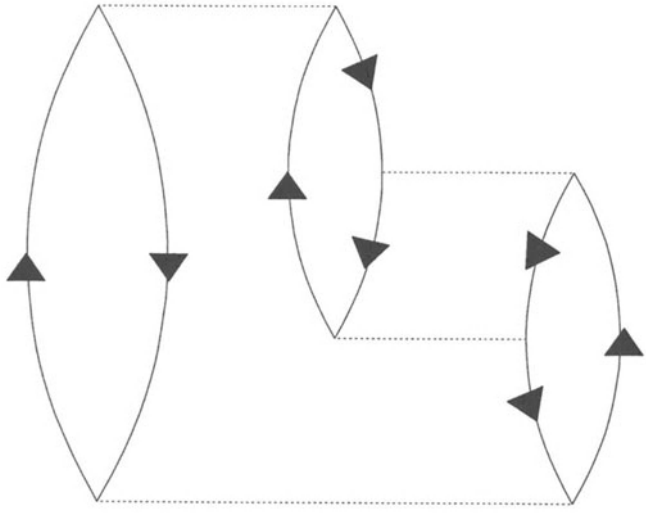


Figure 21. A fourth-order diagram in the many-body perturbation theory series for the correlation energy of a closed shell system involving a triply excited intermediate state.

The second-order energy represents the dominant contribution to the electron correlation energy. For systems that are well described in zero order by a Hartree–Fock reference function—for example, the argon atom[42]—it is known that E_2 accounts for in excess of 99% of the empirical estimate of the correlation energy. For systems such as the beryllium atom in which there are substantial nondynamical correlation effects, E_2 may account for only 80% of the correlation energy.[43]

Explicitly carrying out the spin integrations in the above equation, we obtain

$$\begin{aligned}\langle pq|\,0\,|rs\rangle &= \langle pq|\,r_{12}^{-1}\,|rs\rangle - \langle pq|\,r_{12}^{-1}\,|sr\rangle \\ &= \langle PQ|\,r_{12}^{-1}\,|RS\rangle\,\delta(\sigma_p,\sigma_r)\,\delta(\sigma_q,\sigma_s) \\ &\quad - \langle PQ|\,r_{12}^{-1}\,|SR\rangle\,\delta(\sigma_p,\sigma_s)\,\delta(\sigma_q,\sigma_r)\end{aligned}$$

where we use upper case indices P, Q, R, ... to denote arbitrary (spatial) orbitals and σ_p is the spin function associated with spin-orbitals p. Thus the second-order energy takes the form

$$\begin{aligned}E_2 = \tfrac{1}{4}\sum_{IJ}\sum_{AB}\{&([IA|JB]-[IB|JA])([AI|BJ]-[AJ|BI]) \\ &+[IA|JB][AI|BJ]+[IB|JA][AJ|BI]\}(\varepsilon_I+\varepsilon_J-\varepsilon_A-\varepsilon_B)^{-1}\end{aligned}$$

where we use I, J, K, ... for occupied orbitals and A, B, C, for unoccupied orbitals. In the above equation, we have employed the usual charge-cloud

notation; that is, $[PR|QS] = \langle PQ| r_{12}^{-1} |RS\rangle$. It should be noted that only integrals of the type

$$[IA|JB]$$

arise in the calculation, and, therefore, only a partial transformation of two-electron integrals is required prior to the second-order energy calculation.

The second-order energy, as is well known, may, therefore, be written as a sum of pair energies

$$E_2 = \sum_{I \geqslant J} e_{IJ}$$

with

$$e_{IJ} = \{1 + [1 - \delta(I, J)]\} \sum_{AB} \{2[IA|JB]^2 - [IA|JB][IB|JA]\}(\varepsilon_I + \varepsilon_J - \varepsilon_A - \varepsilon_B)^{-1}$$

The last two equations define the correlation energy component whose evaluation we consider in this paper.

A control flow graph for the evaluation of the second-order many-body perturbation theory energy is given in Figure 21.

To evaluate the expression for e_{IJ} efficiently on a vector processor, we define the vector intermediates

$$T1(P) = [IA|JB], \qquad P = An_{\text{unocc}} + B, \qquad A, B = 1, 2, ..., n_{\text{unocc}}$$

where n_{unocc} is the number of unoccupied orbitals,

$$T2(P) = [IB|JA], \qquad P = An_{\text{unocc}} + B, \qquad A, B = 1, 2, ..., n_{\text{unocc}}$$

and

$$D(P) = \varepsilon_I + \varepsilon_J - \varepsilon_A - \varepsilon_B, \qquad P = An_{\text{unocc}} + B, \qquad A, B = 1, 2, ..., n_{\text{unocc}}$$

The evaluation of e_{IJ} then involves the following vector operators:

```
T3(P) = T1(P) * T1(P), for all P
T4(P) = T1(P) * T2(P), for all P
T5(P) = T3(P) + T3(P) - T4(P), for all P
T6(P) = T5(P)/D(P), for all P
E     = SUM(T6)
```

where SUM(X) evaluates the sum of the elements of the vector X, and E

is the contribution to the second-energy energy for the pair (I, J). The FORTRAN code that we found to perform these operations most efficiently on the Cyber 205 or a single processor ETA-10 is given in Figure 23.

Now the code given in Figure 23 is suitable for direct many-body perturbation theory calculations[37, 44] in which the two-electron integrals over molecular orbitals are generated as they are required. This is an extension of the direct self-consistent field method proposed by Almlof, Faegri, and Korsell.[45] Such an approach is particularly attractive for calculations on large molecules since it avoids the need to store large numbers of integrals. However, more usually the two-electron integrals are read from a disk together with the corresponding labels, and the unpacking of these labels is a significant computational task.[44]

To evaluate the second-order energy in a multiprocessing environment, we took the approach of evaluating the pair correlation energy associated with different electron pairs on different processors concurrently. This is a macrotasking as opposed to a microtasking approach as is favored because the computations for different pairs can essentially proceed independently. The crucial aspect of this approach is to ensure that the load is balanced, that is, that no processor is idle for a significant period of time. Now the calculation of the second-order energy component involves the evaluation of

$$\tfrac{1}{2}n_{\text{occ}}(n_{\text{occ}}+1)$$

pair correlation energies and the number of these will, except for very small systems, always be considerably less than the number of processors available. Thus in the worst case we may have only one processor active for the time period required to evaluate a single pair energy. When the number of processors is very much less than the number of electron pairs the calculation will be performed M times faster on an M-processor computer than it would be performed on a single processor.

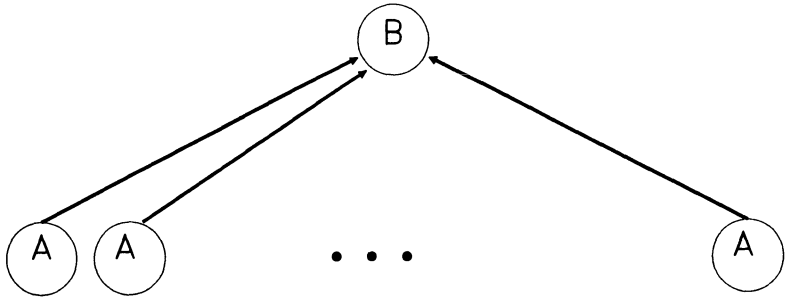


Figure 22. A control flow graph for second-order many-body perturbation theory calculations. A represents the computation of a single pair energy; B the summation of these pair energies to give the total second-order energy.

FORTRAN code written to evaluate the second-order energy on a multitasking computer, the CRAY X-MP/48, is given in Figure 24.

Since we wished to perform timing tests for a wide range of n_{occ}, the number of occupied orbitals, and n_{unocc}, the number of unoccupied orbitals, we avoided the need to execute the stages of an electronic structure calculation which normally precede a many-body perturbation theory calculation by employing dummy two-electron integrals, that is, we employed a set of arbitrary floating point numbers having all the symmetry properties of the actual two-electron integrals.

```
      SUBROUTINE E2CAL(NOCC,NORB)
*
*     Second-order many-body perturbation theory
*
*     Version optimized for the CDC Cyber 205 and single processor
*     ETA-10 machines.
*
*     Language : FORTRAN 200.
*
*     Note the use of special calls.
*
      COMMON /DATA1/ EORB(1010),DIJA(1010)
      COMMON /DATA2/ T1(1000000),T2(1000000),D(1000000)
*
*     Input :-
*
*     Scalars
*
*        NOCC          Number of occupied orbitals
*
*        NORB          Total number of orbitals
*
*
*     Arrays
*
*           EORB(*)    Orbital energies
*
*           T1(*)      Integrals defined in equation (12)
*
*           T2(*)      Integrals defined in equation (13)
*
*     Work arrays
*
*           DIJA(*)    Denominator intermediate.
*
*           D(*)       Pair energy intermediate.
*
```

Figure 23. Second-order many-body perturbation theory FORTRAN code for a vector processor, the Cyber 205 of a single processor ETA-10 machine (taken from Ref. 37).

```
*     Executable statements
*
      N1 = NOCC + 1
      NVIRT = NORB - NOCC
      NCOUNT = NVIRT * NVIRT
      SUM = 0.0
*
*     Sum over pair energies
*
      DO 1 I=1,NOCC
         DO 2 J=1,I
            DIJ = EORB(I) + EORB(J)
            SUBSUM = 0.0
            NP = 1
            DIJA(1;NVIRT) = DIJ - EORB(1;NVIRT)
         DO 3 M1 = 0,NCOUNT-1,65535
            M2 = MINO(NCOUNT-M1,NVIRT)
            T2(M1+1;M2) = 2.0*T1(M1+1;M2)-T2(M1+1;M2)
 3       CONTINUE
         DO 4 IA=1,NVIRT
            D(NP;NVIRT) = (T1(NP;NVIRT) / (DIJA(IA)
               EORB(N1;NVIRT))) * T2(NP;NVIRT)
            NP = NP + NVIRT
 4       CONTINUE
         DO 5 J1=0,NCOUNT-1,65535
            J2 = MINO(NCOUNT-J1,65535)
            SUBSUM = SUBSUM + Q8SSUM(D(J1+1;J2))
 5       CONTINUE
         IF(I.NE.J) SUBSUM = 2.0 * SUBSUM
         SUM = SUM + SUBSUM
 2    CONTINUE
 1  CONTINUE
    RETURN
    END
```

Figure 23. (*Continued*)

The dependence of the rate of computation of the second-order energy on the number of unoccupied orbitals using one processor on the CRAY X-MP/48 computer is shown in Figure 25. These results are taken from the recent work of Moncrieff, Baker, and Wilson.[37] The rate of computation approaches about 152 MFLOPS, which should be compared with a maximum possible rate of about 235 MFLOPS for this machine.

Execution of the FORTRAN code shown in Figure 24 for an increasing number of unoccupied orbitals using four processors leads to a rate of computation approaching 600 MFLOPS. The dependence of the rate of computation on the number of unoccupied orbitals for the CRAY

```
      SUBROUTINE E2CAL(NOCC,NORB,NCPUS)
*
*     Second-order many-body perturbation theory
*
*     Version optimized for the CRAY X-MP/48 multi processor
*     machine.
*
*     Language : CRAY FORTRAN.
*
      COMMON /DATA1/ EORB(2000)
      COMMON /DATA2/ T1(1000000), T2(1000000)
      COMMON /TSK/ TASK, PSUM(100), IVAL(100), JVAL(100)
      COMMON /NO/ MOCC, MORB, N1
*
*     Input :-
*
*     Scalars
*
*        NOCC          Number of occupied orbitals
*
*        NORB          Total number of orbitals
*
*        NCPUS         Number of processors
*
*     Arrays
*
*           EORB(*)    Orbital energies
*
*           T1(*)      Integrals defined in equation (12)
*
*           T2(*)      Integrals defined in equation (13)
*
*     Work arrays
*
*           PSUM(*)       Intermediate.
*
*           IVAL(*)       Intermediate.
*
*           JVAL(*)       Intermediate.
*
*           TASK(3,4)
*
*     Executable statements
*
      INTEGER TASK(3,4)
      EXTERNAL E2MULT
*
      DO 1 I=1,NCPUS
         TASK(1,I)=3
         TASK(3,I)=I
```

Figure 24. Second-order many-body perturbation theory FORTRAN code for a multitasking computer, the CRAY X-MP/48 (taken from Ref. 37).

```
    1    CONTINUE
         CALL TSKTUNE('MAXCPU',NCPUS)
         CALL TSKTUNE('DBRELEAS',NCPUS)
*
*
*
      SUM=0.0
      K=0
      K2=0
      MOCC = NOCC
      MORB = NORB
      N1=1+NOCC
  *
  *     Sum over pair energies
  *
      DO 2 II=1,NOCC
         DO 3 JJ=1,II
         IVAL(II)=II
         JVAL(JJ)=JJ
         K=K+1
         K2=K2+1
         IF(K.NE.NCPUS) THEN
            CALL TSKSTART(TASK(1,K),E2MULT,IVAL(II),JVAL(JJ),PSUM(K2))
         ELSE
            CALL E2MULT(IVAL(II),JVAL(JJ),PSUM(K2))
            DO 4 KK=1,NCPUS-1
               CALL TSKWAIT(TASK(1,KK))
    4       CONTINUE
            K=0
         ENDIF
    3    CONTINUE
    2 CONTINUE
      DO 5 KK=1,K
         CALL TSKWAIT(TASK(1,KK))
    5 CONTINUE
      DO 6 MM=1,K2
         SUM=SUM+PSUM(MM)
    6 CONTINUE
      RETURN
      END
      SUBROUTINE E2MULT(II,JJ,TOTAL)
*
*     Evaluate the energy for the pair (i,j)
*
      DIMENSION D(1010)
      COMMON /DATA1/ EORB(2000)
      COMMON /DATA2/ T1(1000000), T2(1000000)
      COMMON /NO/ NOCC, NORB, N1
  *
  *     Executable statements
  *
```

Figure 24. (*Continued*)

```
      DIJ = EORB(II) + EORB(JJ)
      IP = 0
      TOTAL = 0.0
      DO 1 IA=N1,NORB
         DIJA = DIJ - EORB(IA)
         DO 2 IB=N1,NORB
            IP=IP+1
            D(IB) = T1(IP) / (DIJA - EORB(IB))
            TOTAL = TOTAL + D(IB) * (2.0 * T1(IP) - T2(IP))
    2    CONTINUE
    1 CONTINUE
      IF(II.NE.JJ) TOTAL = 2.0 * TOTAL
      RETURN
      END
```

Figure 24. (*Continued*)

X-MP/48 (4 processors) computer is shown in Figure 26. These results are also taken from the recent work of Moncrieff, Baker, and Wilson.[37]

Much research will be devoted over the next few years to handling in routine electronic structure studies molecular systems containing larger numbers of electrons than has been possible to date. The revolution in computer architectures that has been taking place over the past decade, particularly the advent of vector-processing machines and very powerful machines that allow for the organization of concurrent computation, is leading to something of a revolution in the size and accuracy of molecular electronic structure studies. Indeed, some authors have commented that such calculations will be dominated more and more by the available computational resources.

5.2.4. *Example 4: Relativistic Molecular Electronic Structure Calculations*

In Figure 27, we classify the types of molecular electronic structure calculations that are becoming susceptible to computational investigation with the increasing power of computing machines. Region A corresponds to the "traditional" area for electronic structure studies—molecules containing smaller numbers of both atoms and electrons per atom. Region B, representing systems containing larger numbers of atoms but with the number of electrons per atom still relatively small, is being attacked by a combination of powerful computing devices and modern "many-body" theories of the type described in Section 5.2.3. Region C is also attracting interest. This represents systems containing a smaller number of atoms but with on average a larger number of electrons per atom. The description of

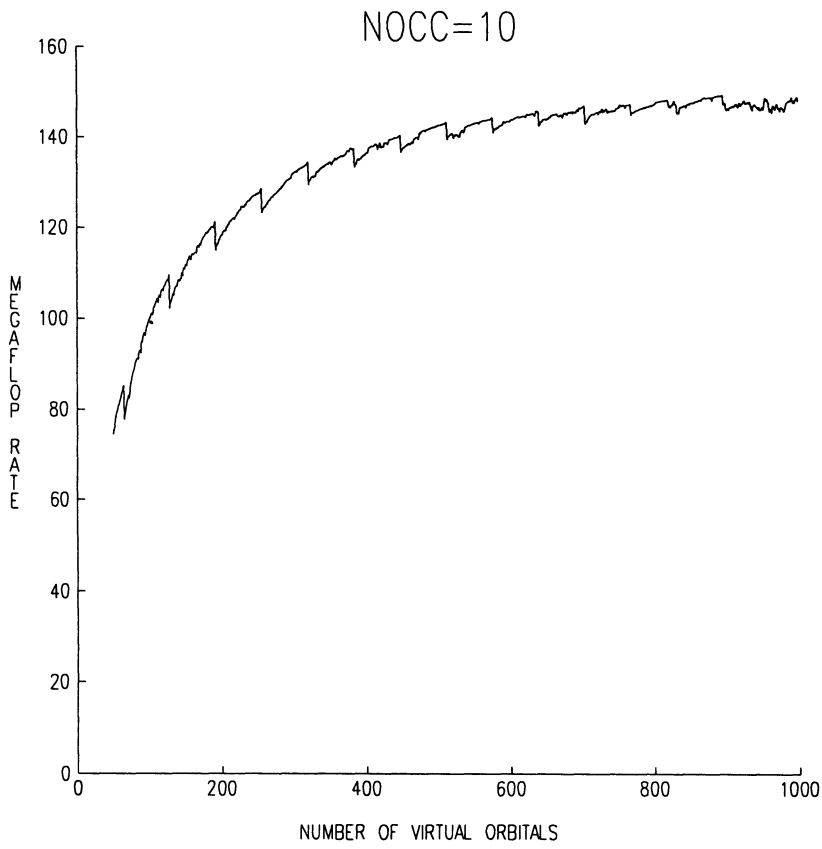


Figure 25. Dependence of the rate of computation on the number of unoccupied orbitals for the CRAY X-MP/48 (1 processor) computer (taken from Ref. 37).

the electronic structure of molecular systems that fall into this category demands a treatment of relativistic effects.

In recent years, there has been a growing interest[46–51] in the development of quantum chemical techniques based on relativistic quantum mechanics. For systems that contain heavy atoms nonrelativistic quantum mechanics is inadequate because the mean speed of the core electrons is a substantial fraction of the speed of light, so that a fully relativistic electronic structure theory is required, both at the independent particle level and beyond. As we have recently observed[51]

> If one were to hazard a guess as to where the major new developments in quantum chemistry will be in the next fifteen years, the proper treatment of relativity and the introduction of quantum electrodynamics effects seems to us to be a likely candidate, since these may be more important than electron correlation in heavy elements, and there is no evidence that their effects are simply additive.

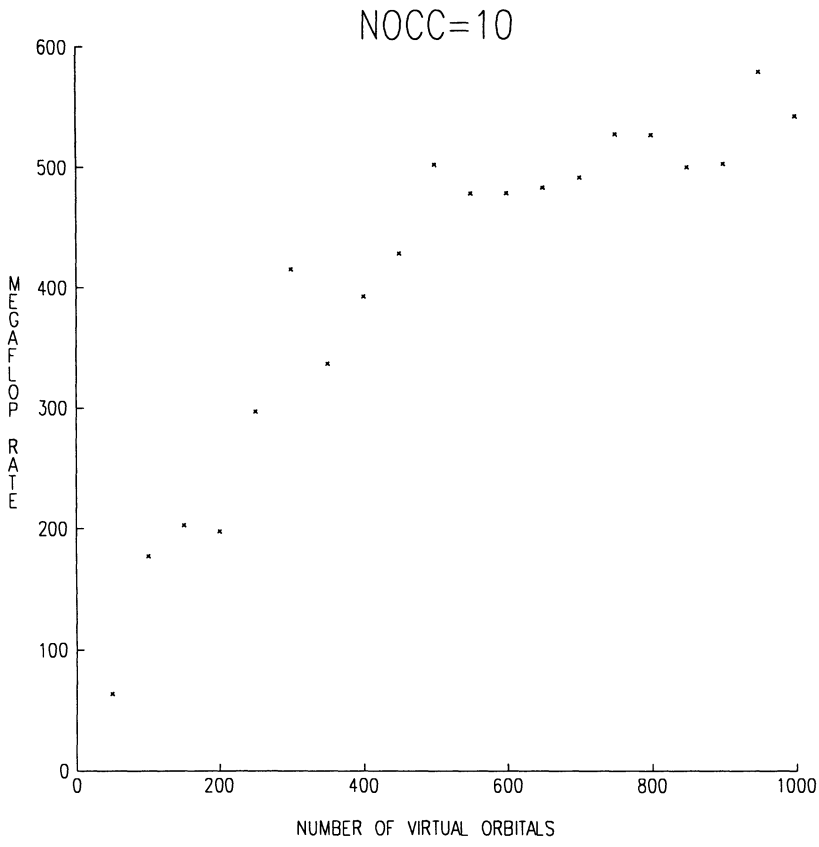


Figure 26. Dependence of the rate of computation on the number of unoccupied orbitals for the CRAY X-MP/48 (4 processor) computer (taken from Ref. 37).

Indeed, there is a tacit assumption made in virtually every published molecular electronic structure calculation, even those that purport to be "*ab initio*": the physical consequences of the special theory of relativity are completely negligible, except under the most extreme conditions. The further implication is that should such effects be observed experimentally, they constitute a well-defined perturbation, to be appended to a standard nonrelativistic calculation.

The treatment of the relativistic electronic structure problem raises a number of fundamental theoretical problems as well as significant computational demands. A detailed description of the theoretical aspects of the problem lies outside the scope of the present chapter, and we shall only mention them insofar as they are necessary to gain an understanding of the magnitude of the computational aspects of the problem. Volume 2 in the

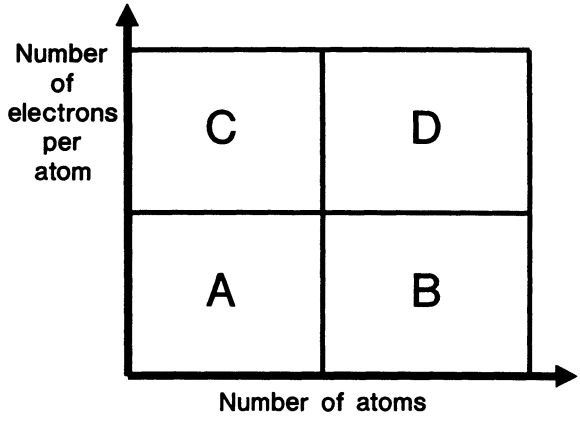


Figure 27. Type of molecular electronic structure calculations that are becoming susceptible to computational investigation with the increasing power of computing machines. Region A defines the type of molecule that has traditionally been studied in computational quantum chemistry, a system containing a small number of light atoms. Regions B and C defines the types of molecules for which computational studies of the electronic structure have become tractable by exploiting contemporary supercomputers. Region B represents extended molecules containing light atoms, while region C represents molecules containing fewer but heavier atoms. Region D, representing extended molecular systems containing heavy atoms, will certainly provide a challenge for future supercomputers.

present series provides a detailed overview of the theoretical aspects of the problem.

Putting aside the problem of an appropriate choice of relativistic Hamiltonian, a problem that has no parallel in nonrelativistic electronic structure theory, a major problem arises when, as is almost obligatory in molecular calculations, we invoke the algebraic approximation and expand the large and small components of the relativistic wave function in some finite analytic basis sets. To gain a qualitative understanding of the problem we refer to Figure 28, in which a comparison of the Dirac and the more familiar Schrödinger spectra for point nuclear hydrogenic ions is made. The Dirac spectrum consists of positive and negative energy branches, both of which must be considered if a more complete description of the system is required. If the basis sets are not chosen with considerable care (we refer the interested reader to Ref. 51 for full details) we can easily fail to get a clean separation of the finite basis set representation of the two branches, which undermines the basis of the relativistic quantum theory of electronic structure.

Once appropriate basis sets have been constructed, relativistic self-consistent field calculations can be carried out in a way that is analogous to

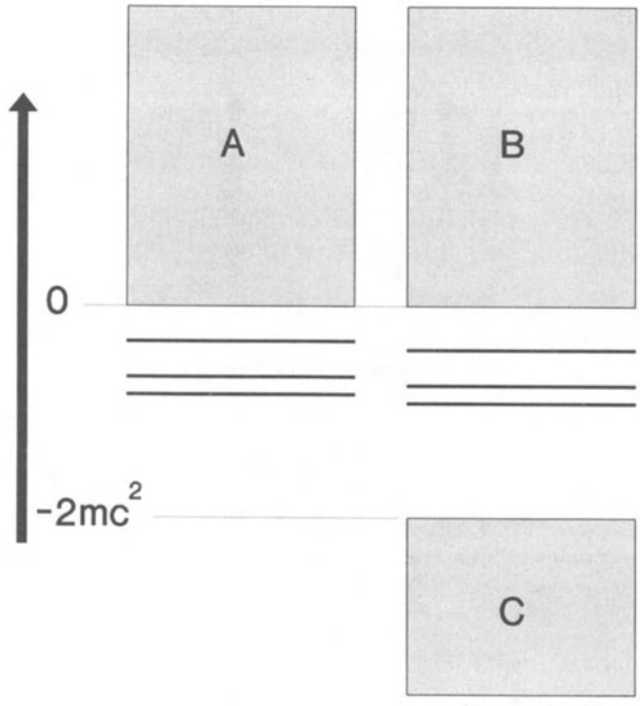


Figure 28. Comparison of the Dirac and the Schrödinger spectra for point nuclear hydrogenic ions. The Schrödinger spectrum, displayed on the left, contains a continuum A. The Dirac spectrum, displayed on the right, contains a "positive energy" continuum B and a "negative energy" continuum C.

the nonrelativistic case. Some typical atomic results are shown in Figure 29, where the shift in the orbital energies of the mercury Dirac–Hartree–Fock ground state due to relativity is indicated. The shift is defined to be $[E(\mathrm{r}) - E(\mathrm{nr})]/E(\mathrm{nr})$, where $E(\mathrm{r})$ is a relativistic orbital energy and $E(\mathrm{nr})$ is the corresponding nonrelativistic quantity. These results, due to the finite basis set program SWIRLES, are in full agreement with those from finite difference calculations (taken from the work of Quiney, Grant and Wilson[51]).

Relativistic calculations can be performed in a similar fashion for molecules. In Figure 30, the shift in the orbital energies of the nitrogen molecule Dirac–Hartree–Fock ground state due to relativity are shown. The shift is defined to be $[E(\mathrm{r}) - E(\mathrm{nr})]/E(\mathrm{nr})$, where $E(\mathrm{r})$ is a relativistic orbital energy and $E(\mathrm{nr})$ is the corresponding nonrelativistic quantity (taken from the work of Quiney, Grant, and Wilson[51]). These calculations

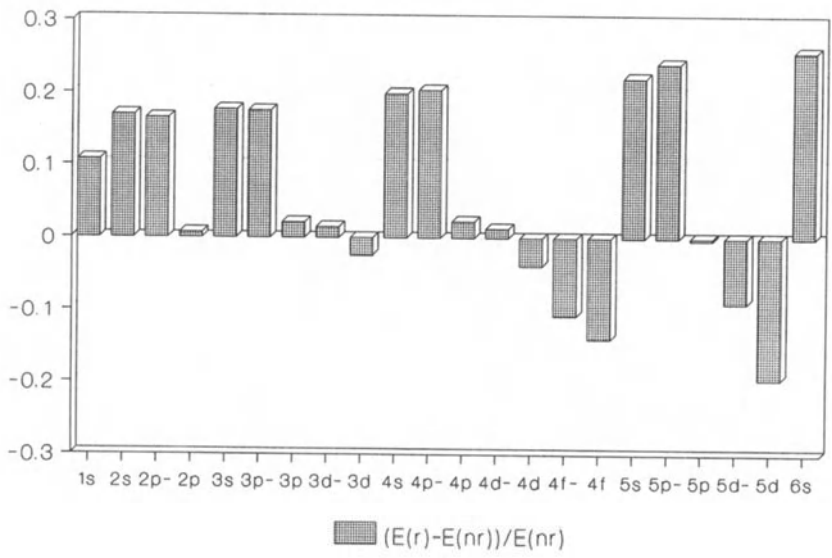


Figure 29. The shift in the orbital energies of the mercury Dirac–Hartree–Fock ground state due to relativity. The shift is defined to be $[E(\mathrm{r}) - E(\mathrm{nr})]/E(\mathrm{nr})$, where $E(\mathrm{r})$ is a relativistic orbital energy, and $E(\mathrm{nr})$ is the corresponding nonrelativistic quantity. These results, due to the finite basis set program SWIRLES, are in full agreement with those from finite difference calculations. (Taken from Ref. 51.)

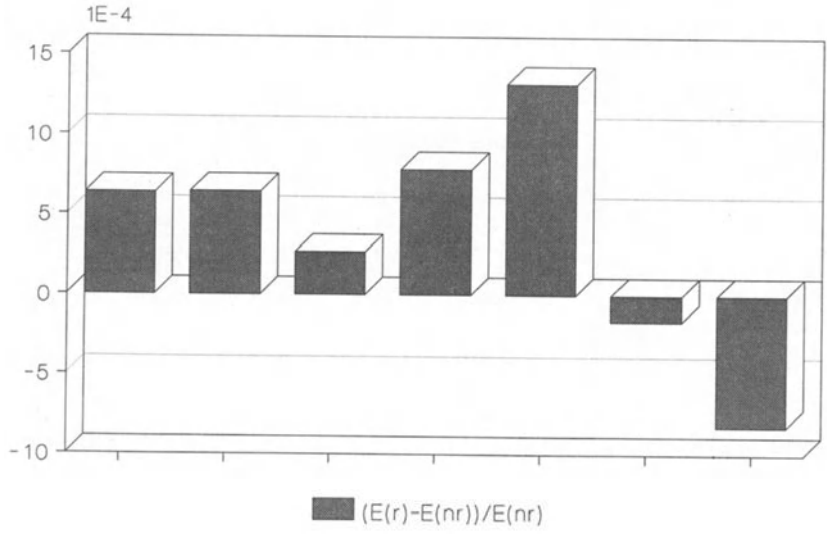


Figure 30. The shift in the orbital energies of the nitrogen molecule Dirac–Hartree–Fock ground state due to relativity. The shift is defined to be $[E(\mathrm{r}) - E(\mathrm{nr})]/E(\mathrm{nr})$, where $E(\mathrm{r})$ is a relativistic orbital energy and $E(\mathrm{nr})$ is the corresponding nonrelativistic quantity. (Taken from Ref. 51.)

were performed with the relativistic electronic structure program RATMOL.[49, 52]

To give some idea of the magnitude of the computational demands of fully relativistic self-consistent field calculations, we display schematically the structure of the Dirac–Fock matrix in Figure 31. Whereas the nonrelativistic Fock matrix is of dimension $N \times N$, where N is the size of the basis set, the Dirac–Fock matrix is of dimension $4N \times 4N$. Not only is the construction of the Dirac–Fock matrix more demanding than the nonrelativistic Fock matrix, but, remembering that matrix diagonalization scales as the third power of the dimension, the amount of computation increases by a factor of 64 in the relativistic case.

When we turn to the relativistic electron correlation problem, the many-body perturbation theory becomes a particularly appropriate approach. In the nonrelativistic theory, electron correlation effects are described by diagrams depicting various excitations of electrons from below the Fermi level, creating particles above it and holes below it. In the relativistic theory, excitations from the negative energy branch of the spectrum are possible. The negative energy "sea" is taken to be filled with an infinite number of electrons excitation of which creates virtual electron–positron pairs. By way of example, in Figure 32, the second-order energy diagrams for the relativistic electron correlation problem in a closed-shell system are displayed. The diagrams displayed involve (a) no virtual pair creation—these diagrams are entirely equivalent to those which arise in nonrelativistic theory and involve only states that arise in the positive energy branch of the spectrum, (b) and (c) the creation of one virtual electron–positron pair, and (d) the creation of two virtual pairs. The

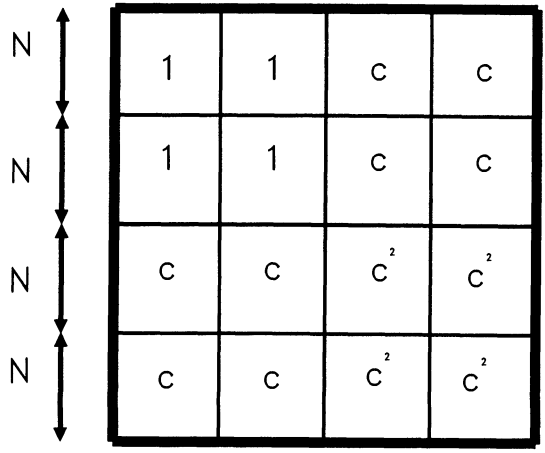


Figure 31. The structure of the Dirac–Fock matrix. The sixteen subblocks of the Dirac–Fock matrix are either of orders unity, c or c^2, where c is the speed of light.

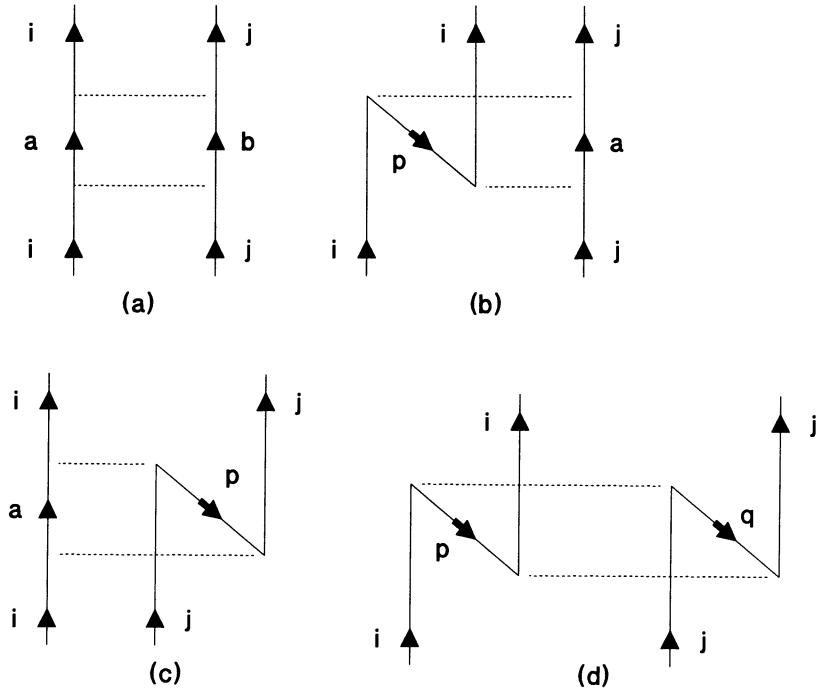


Figure 32. Second-order energy diagrams for the relativistic electron correlation problem involving (a) no virtual pair creation, (b) and (c) the creation of one virtual pair, and (d) the creation of two virtual pairs. The indices *i, j, k,* ... are used to label occupied positive energy state, *a, b, c,* ... label unoccupied positive energy states and *p, q, r,* ... label negative energy states. (Taken from Ref. 51.)

indices *i, j, k,* ... are used to label occupied positive energy state, *a, b, c,* ... label unoccupied positive energy states and *p, q, r,* ... label negative energy states (taken from the work of Quiney, Grant, and Wilson[51]). The algebraic expressions corresponding to these diagrams are easily written down. In Figure 33, the second-order energy components for heliumlike system are shown. In particular, the dependence of the following energy components on nuclear charge is indicated:

(a) $E_2(++)$ for excitations $1s^2_{1/2} \rightarrow ns_{1/2}ns_{1/2}$

(b) $E_2(--)$ for excitations $1s^2_{1/2} \rightarrow ns_{1/2}ns_{1/2}$

(c) $E_2(++)$ for excitations $1s^2_{1/2} \rightarrow np_{1/2}np_{1/2}$

(d) $E_2(--)$ for excitations $1s^2_{1/2} \rightarrow np_{1/2}np_{1/2}$

(taken from the work of Quiney, Grant, and Wilson[51]).

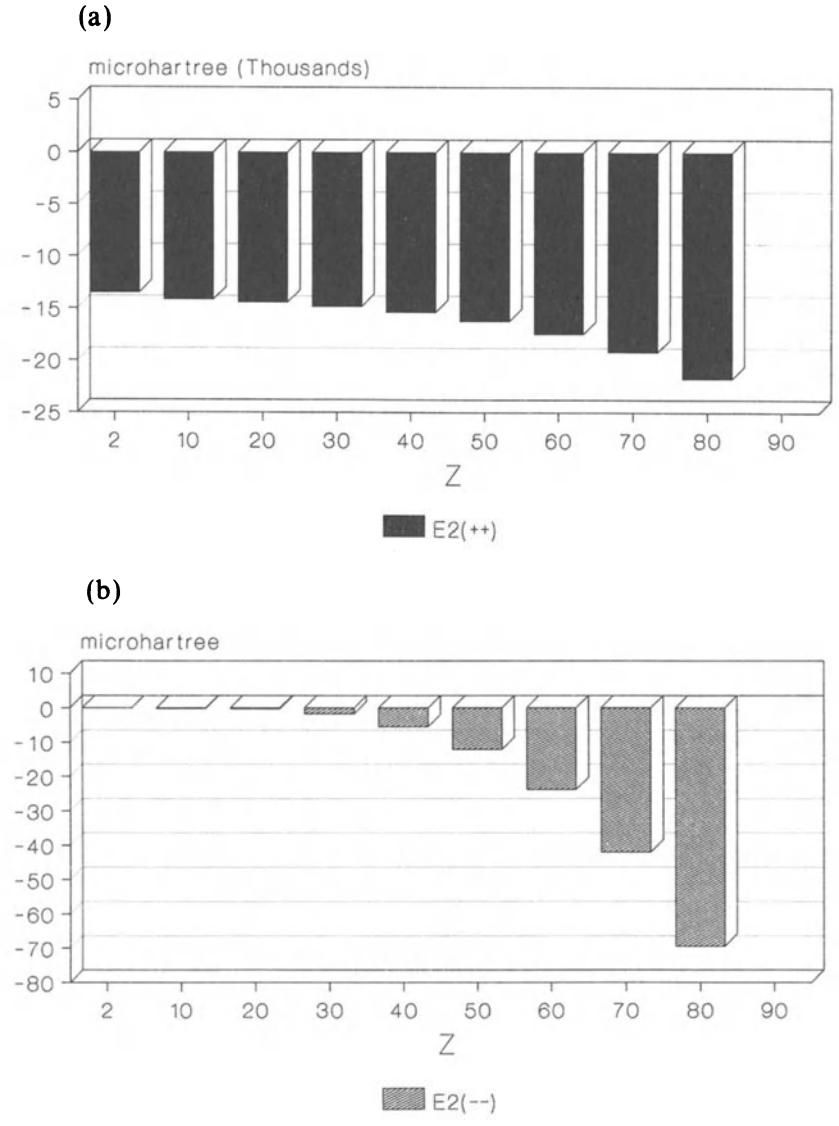
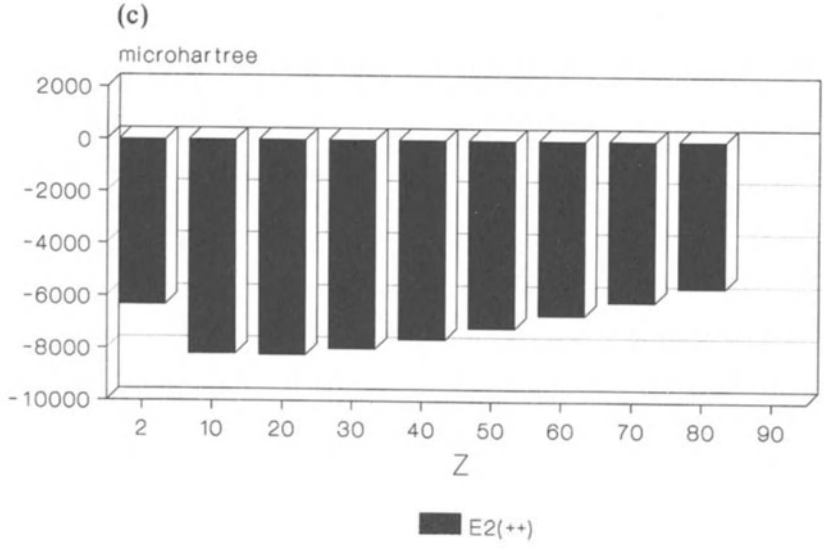
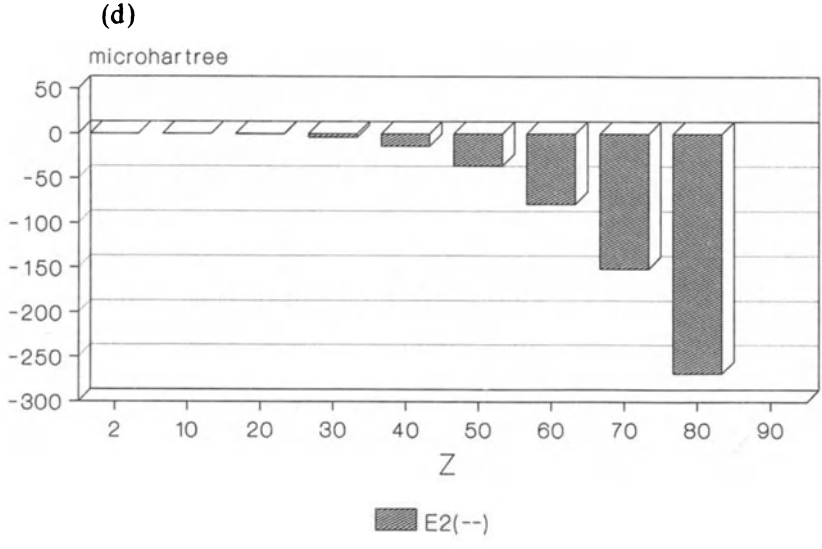


Figure 33. Second-order energy components for heliumlike system. Dependence of the following energy components on nuclear charge: (a) $E_2(++)$ for excitations $1s^2_{1/2} \rightarrow ns_{1/2}ns_{1/2}$;





(b) $E_2(--)$ for excitations $1s_{1/2}^2 \rightarrow ns_{1/2}ns_{1/2}$; (c) $E_2(++)$ for excitations $1s_{1/2}^2 \rightarrow np_{1/2}np_{1/2}$; (d) $E_2(--)$ for excitations $1s_{1/2}^2 \rightarrow np_{1/2}np_{1/2}$. (Taken from Ref. 51.)

Although this development is due to fundamental theoretical advances, it has had to wait for the availability of computing machines with sufficient power. In 1967, one of Roothaan's students, Y.-K. Kim, wrote, in his pioneering work on the use of basis set expansion methods in relativistic atomic structure work[53].

> At present the outlook for the application of our method to large atoms ($Z > 50$) or molecules is not bright. The computers currently available are too slow and short on memory to handle the large number of basis functions needed to represent the Hartree–Fock solutions to an accuracy which make relativistic calculations meaningful.

From the above discussion the computational demands of the relativistic self-consistent field and electron correlation problem, which includes the four-index transformation should be apparent.

Finally, we return to Figure 27, in which we have classified the types of molecular electronic structure calculations that are becoming susceptible to computational investigation with the increasing power of computing machines. Region A is the area of "traditional" activity in the molecular electronic structure field. Regions B and C have been discussed above. It is in region D, representing systems containing large numbers of atoms with, on average, large numbers of electrons per atom, that the main challenge in electronic structure calculations will come in the decades ahead. It is not at all difficult to think of problems that would overwhelm any computing machine that is likely to become available before the end of this century.

6. Prospects and Concluding Remarks

It is clear that with the availability of ever more powerful computational resources many areas of computational chemistry will continue to grow and flourish. New areas of application will also appear. We are already seeing the application of the methods of computational quantum chemistry to larger molecules than has previously been possible leading to challenging new areas of application. At the same time, by devising more sophisticated theories of electronic structure together with efficient algorithms tailored to modern high-performance computers, the properties of smaller molecular systems are being computed to a higher precision than has previously been possible. Areas that involve substantial new physics are also opening up—areas such as the breakdown of the Born–Oppenheimer approximation, an approximation that is fundamental to the vast majority of quantum chemical calculations performed to date. The importance of relativistic effects in the chemistry of the heavy (and even not so heavy) elements is becoming increasingly recognized and, as we have indicated in the discussion above, involve some quite fundamental new science.

Molecular mechanics has extended the range of molecules that can be studied by computational methods, not only can proteins be modeled in isolation but proteins in aqueous solution and the interactions between proteins can be investigated. The usefulness of the molecular mechanics approach is determined to a large extent by the accuracy of the parametrization. Clearly, quantum chemical studies will play an increasingly important role here. Equally, in molecular dynamics and Monte Carlo simulations and in molecular collision theory the availability of increasingly accurate potential energy functions will lead to new advances in the accuracy and sophistication of calculations. A knowledge of the form of the potential energy surfaces is fundamental to the theoretical study of chemical reactions. In this respect, it is perhaps interesting to reiterate some remarks made by Krauss[54] almost twenty years ago:

> It is frequently noted that the availability of high-speed digital computers now permits the calculation of adiabatic potential energy surfaces. After noting that fact, attention is then turned to the difficult scattering problem attendant upon the use of these surfaces. It is therefore somewhat disconcerting to survey the reality of molecular surfaces. Notwithstanding the very considerable recent progress, the number of problems solved by these calculations is meager...

Of course, considerable progress has been made in the past 20 years; both on the theoretical and the computational aspects of the problem. However, we note the comment made about ten years ago by Connors[55] on the accuracy required of *ab initio* calculations of potential energy curves and surfaces:

> 1 kcal mol^{-1} is often regarded as "chemical accuracy"; this is not necessarily true for scattering calculations. It is known, for example, that a small change in a potential surface can produce considerably different scattering results.

It appears that, even today, although much progress has been made, there is still a great deal to be done.

Other fields of computational chemistry—the design of organic synthetic routes, the solution of the equations governing complex chemical kinetics problems and molecular graphics—will all benefit enormously from increasingly powerful computational resources.

We conclude this overview of concurrent computation in chemistry and the chemical sciences by making two remarks about the future. The first remark concerns the nature of the computers used by chemists and the second, perhaps more important, remark addresses the role of computation in chemistry and, indeed, science in general.

To date, the vast majority of computational chemists have been content to use general purpose computers in their work designing software to exploit the capabilities of commercially available machines. Ostlund[56] has

expressed the view that they should be involved in the construction of the machines that they use:

> Computational chemistry is too established and important a field to leave its only significant piece of apparatus to disinterested computer scientists. Chemists in other fields have commonly built very sophisticated instruments—molecular beam machines, electron scattering spectrometers, ion cyclotron resonance spectrometers, etc. ... If computational chemists are to take full advantage of the current revolution in microelectronics, which offers great prospects for cost-effective high-performance computation, it will be necessary to become involved in the design and building of highly parallel architectures specific to specific chemical applications.

To some extent some computational chemists are already designing their own computer systems by buying commercially available components and thus tailoring their complete system to their specific needs. For example, a minicomputer system used for quantum chemical calculations may contain more disk storage than would be required by the average user. At present economics dictates against this approach for state-of-the-art supercomputers, which, because of their cost, have to be shared between many users from many different disciplines.

Finally, we turn to the more general question of the purpose of the computational approach in chemistry. We recall the view expressed nearly 40 years ago by E. P. Wigner on the role of computation in the physical sciences[57]:

> If I had a great calculating machine, I would perhaps apply it to the Schrödinger equation of each metal and obtain its cohesive energy, its lattice constant, etc. It is not clear, however, that I would gain a great deal by this. Presumably, all the results would agree with the experimental values and not much would be learned from the calculation. What would be preferable, instead, would be a vivid picture of the behavior of the wave function, a simple description of the essence of metallic cohesion, and an understanding of the causes of it variation from element to element. Hence the task which is before us is not a purely scientific one, it is partly pedagogic. Nor can its solution be unique: the same wave function can be depicted in a variety of ways (just as the cubic close-pack lattice can), the same energy can be decomposed in a variety of ways into different basic constituents. Hence the value of any contribution will depend on the taste of the reader. In fact, from the point of view of the present article, the principal purpose of accurate calculations is to assure us that nothing truly significant has been overlooked.

This point of view was not uncommon among physical scientists in the precomputer and early computer days when calculations were necessarily of a more qualitative nature. Nowadays, in quantum chemistry, for example, quantitative accuracy can frequently be achieved in molecular electronic structure studies and, to some extent, the simple concepts that emerged from the pioneering work of the first half of the century are often put to one side. Much will be gained, however, if the computational

chemist can devise theories and associated computational algorithms that can not only afford high accuracy but also insight into the problem at hand.

Acknowledgment

I wish to thank Dr. David Moncrieff for providing the timing data for the ETA 10 computer, Dr. Harry M. Quiney for useful discussions on the relativistic electronic structure problem, and V. R. Saunders for advice on multitasking on the CRAY X-MP/48.

References

1. Future Facilities for Advanced Research Computing. The report of a Joint Working Party, June 1985 (Advisory Board for the Research Councils, Computer Board for Universities and Research Councils, University Grants Committee), SERC, 1985.
2. S. Wilson, *Chemistry by Computer, An overview of the applications of computers in chemistry*, Plenum, New York (1986).
3. D. M. Silver and K. Ruedenberg, *J. Chem. Phys.* **49**, 4301 (1968).
4. D. M. Silver and K. Ruedenberg, *J. Chem. Phys.* **49**, 4306 (1968).
5. R. E. Christofferson and K. Ruedenberg, *J. Chem. Phys.* **49**, 4285 (1968).
6. E. L. Mehler and K. Ruedenberg, *J. Chem. Phys.* **50**, 2575 (1969).
7. C. A. Coulson, *Proc. Cambridge Philos. Soc.* **34**, 204 (1938).
8. R. McWeeny, Foreword, *Methods in Computational Chemistry*, Plenum Press, **1**, New York (1987).
9. R. G. Parr and B. L. Crawford, National Academy of Sciences Conference on Quantum Mechanical Methods in Valence Theory, *Proc. Natl. Acad. Sci. U.S.* **38**, 547 (1952).
10. H. F. Schaefer III, *Quantum Chemistry, The development of ab initio methods in molecular electronic structure theory*, Clarendon Press, Oxford (1984).
11. P. Lykos and I. Shavitt, *Supercomputers in Chemistry*, American Chemical Society, Washington D.C., 1981.
12. P. Lykos, Computers in the world of chemistry, *Adv. Comput.* **21**, 275 (1982).
13. V. R. Saunders and J. van Lenthe, The direct CI method. A detailed analysis, *Molec. Phys.* **48**, 923 (1983).
14. J. J. Dongarra, Performance of various computers using standard linear equations software in a FORTRAN environment, Technical Memorandum No. 23, Mathematics and Computer Science Division, Argonne National Laboratory (1988).
15. R. McWeeny and B. T. Pickup, Quantum theory of molecular electronic structure, *Rep. Progr. Phys.* **43**, 1065 (1980).
16. S. Wilson, *Electron Correlation in Molecules*, Clarendon Press, Oxford (1984).
17. R. W. Hockney and C. J. Jesshope, *Parallel Computers*, Adam Hilger, Bristol (1981).
18. H. T. Kung, *J. ACM* **23**, 252 (1979).
19. J. J. Dongarra, D. C. Sarensen, K. Connolly, and J. Patterson, Programming methodology and performance issues for advanced computer architecture, *Parallel Comput.* **8**, 41 (1988).

20. M. J. Flynn, Some computer organizations and their effectiveness, *I.E.E.E. Trans. Comput. C* **21**, 948 (1972).
21. R. W. Hockney, MIMD computing in the USA—1984, *Parallel Comput.* **2**, 119 (1988).
22. J. T. Schwartz, *ACM Trans. Prog. Lang. Syst.* **2**, 484 (1980).
23. V. R. Saunders, Molecular integrals for Gaussian-type functions, in *Methods in Computational Molecular Physics* (G. H. F. Diercksen & S. Wilson eds.), Reidel, Dordrecht (1983).
24. M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, New York (1965).
25. M. Metcalfe and J. Reid, FORTRAN 8X, MacMillan, New York (1988).
26. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Software* **5**, 308 (1979).
27. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Algorithm 539: Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Software* **5**, 324 (1979).
28. J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. Hanson, An extended set of FORTRAN basic linear algebra subprograms: Model implementation and test programs, Argonne National Laboratory report No. ANL-MCS-TM-81 (1986).
29. J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. Hanson, An extended set of FORTRAN basic linear algebra subprograms, Argonne National Laboratory report No. ANL-MCS-TM-41 (1986).
30. J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. Hammarling, A proposal for a set of level 3 basic linear algebra subprograms, Argonne National Laboratory report No. ANL-MCS-TM-88 (1987).
31. J. J. Du Croz, P. J. D. Mayes, J. Wasniewski, and S. Wilson, Application of level 2 BLAS in the NAG library, *Parallel Comput.* **8**, 345 (1988).
32. K. Jankowski, in *Methods in Computational Chemistry*, Vol. 1, *Electron Correlation in Atoms and Molecules*, Plenum Press, New York (1987), Chap. 1.
33. I. Lindgren and J. Morrison, *Atomic Many-Body Theory* (2nd. Edition), Springer, Berlin (1986).
34. S. Wilson, *The Many-Body Perturbation Theory of Atoms and Molecules*, Adam Hilger, Bristol (1989).
35. M. Urban, I. Cernusak, V. Kello, and J. Noga, in *Methods in Computational Chemistry*, Vol. 1, *Electron Correlation in Atoms and Molecules*, Plenum Press, New York (1987), Chap. 2.
36. S. Wilson, Diagrammatic many-body perturbation theory of atomic and molecular electronic structure, *Comput. Phys. Rep.* **2**, 389 (1985).
37. D. Moncrieff, D. J. Baker, and S. Wilson, Diagrammatic many-body perturbation expansion for atoms and molecules. VI. Experiments in vector processing and parallel processing for second order energy calculations, *Comput. Phys. Commun.* **55**, 31 (1989).
38. S. Wilson and V. R. Saunders, Diagrammatic many-body perturbation expansion for atoms and molecules. V. *Comput. Phys. Commun.* **19**, 293 (1980).
39. M. F. Guest and S. Wilson, in *Supercomputers in Chemistry*, p. 1 (P. Lykos and I. Shavitt, eds.), American Chemical Society, Washington D.C. (1981).
40. S. Wilson, in *Proc. 5th. Seminar on Computational Problems in Quantum Chemistry* (P. Th. van Duijknen and W. C. Nieuwpoort, eds.), Groningen (1981).
41. S. Wilson, in *Methods in Computational Molecular Physics* (G. H. F. Diercksen and S. Wilson, eds.), Reidel, Dordrecht (1983).
42. B. H. Wells and S. Wilson, *J. Phys. B: At. Mol. Phys.* **19**, 2411 (1986).
43. D. M. Silver, S. Wilson, and C. F. Bunge, *Phys. Rev. A* **19**, 1375 (1979).
44. S. Wilson, in *Methods in Computational Chemistry*, Vol. 1, *Electron Correlation in Atoms and Molecules*, Plenum Press, New York (1987), Chap. 3.

45. J. Almlof, K. Faegri, and K. Korsell, Principle for a direct SCF approach to *ab initio* calculations, *J. Comput. Chem.* **3**, 385 (1982).
46. H. M. Quiney, I. P. Grant, and S. Wilson, On the Dirac equation in the algebraic approximation, *Phys. Scr.* **36**, 460 (1987).
47. H. M. Quiney, I. P. Grant, and S. Wilson, On the accuracy of the algebraic approximation in relativistic electronic structure studies, in *Numerical Determination of the Electronic Structure of Atoms, Diatomic and Polyatomic Molecules*, Proc. NATO Advanced Research Workshop, April 1988, Versailles, Reidel, Dordrecht (1989).
48. H. M. Quiney, I. P. Grant, and S. Wilson, On the accuracy of Dirac–Hartree–Fock calculations using analytic basis sets, *J. Phys. B: At. Mol. Opt. Phys.* **22**, L45 (1989).
49. S. Wilson, Relativistic molecular structure calculations, in *Methods in Computational Chemistry*, Vol. 2, *Relativistic Effects in Atoms and Molecules*, Plenum Press, New York (1987), Chap. 2.
50. H. M. Quiney, Relativistic many-body perturbation theory, in *Methods in Computational Chemistry*, Vol. 2, *Relativistic effects in Atoms and Molecules*, Plenum Press, New York (1988), Chap. 5.
51. H. M. Quiney, I. P. Grant, and S. Wilson, On the relativistic many-body perturbation theory of atomic and molecular electronic structure, in *Many-Body Methods in Quantum Chemistry* (U. Kaldor, ed.), Lecture Notes in Chemistry, Springer-Verlag, Berlin (1989).
52. L. Laaksonen, I. P. Grant, and S. Wilson, The Dirac equation in the algebraic approximation. VI. Molecular self-consistent field studies using basis sets of Gaussian-type functions, *J. Phys. B: At. Mol. Opt. Phys.* **21**, 1969 (1988).
53. Y.-K. Kim, Relativistic self-consistent-field theory for closed-shell atoms, *Phys. Rev.* **154**, 17 (1967).
54. M. Krauss, Critical review of *ab initio* energy surface, in *Proceedings of the Conference on Potential Energy Surfaces in Chemistry* (W. A. Lester, ed.), IBM, San Jose, California (1971).
55. J. L. Connors, *Comput. Phys. Commun.* **17**, 117 (1979).
56. N. S. Ostlund, in *Personal Computers in Chemistry*, American Chemical Society, Washington, D.C., 1977.
57. E. P. Wigner, in *Proc. Internat. Confer. Theoret. Phys.*, Science Council of Japan, 1953.

2

Chemical Calculations on Cray Computers

PETER R. TAYLOR, CHARLES W. BAUSCHLICHER, JR., AND DAVID W. SCHWENKE

1. Introduction

The advent of supercomputers has had a profound influence on the development of computational chemistry in the last decade. Any increase in computing power from a given level will, of course, increase the range and size of problems that can be studied, but the influence of supercomputers goes much deeper than this. The need to develop new algorithms to exploit supercomputers fully affects formal mathematical aspects of computational chemistry methodology, while in some areas the ability to perform calculations at new levels of accuracy or on new chemical systems can alter entire computational chemistry strategies. In the present review, our goal is to show how the supercomputers produced by Cray Research, Inc. (CRI) have influenced two areas of computational chemistry—molecular electronic structure and dynamics calculations. We shall discuss aspects of performance and programming for a range of Cray computers, and show how these factors have influenced the methodology and implementation in these

The views, opinions, and/or findings contained in this article are those of the authors and should not be construed as an official NASA position, policy, or decision, unless so designated by other documentation.

PETER R. TAYLOR, CHARLES W. BAUSCHLICHER, JR., AND DAVID W. SCHWENKE • NASA Ames Research Center, Moffett Field, California 94035.

areas. We shall also discuss how the results obtained from calculations have in turn influenced the philosophy behind the calculations.

At the time of writing, several different supercomputer models are available from CRI. All are parallel, pipelined vector computers. The CRAY X-MP series is a development of the original CRAY-1: the machines are equipped with one to four CPUs, up to 16 megawords (MW) of very fast memory with multiple channels to each CPU, and a clock period of 8.5 ns (10 ns on the older "se" series). The X-MP machines are characterized by excellent scalar performance and a rather rapid convergence to their asymptotic performance limit, that is, good performance on arithmetic involving short vectors. For example, they achieve over half their asymptotic performance of 235 million floating-point operations per second (MFLOPS) in multiplication of matrices of order 11×11. The CRAY Y-MP is a natural successor to the X-MP, with a clock period of 6 ns and up to 128 MW of memory. Its performance characteristics are very similar to those of the X-MP. The CRAY-2 is a rather different development, compared to the X-MP, from the original CRAY-1. The clock period is only 4.1 ns, and the machine can be equipped with up to 512 MW of relatively slow memory or up to 128 MW of faster memory. At well over 400 MFLOPS per processor the asymptotic performance is even higher than the Y-MP, but the performance on scalar or short vector arithmetic is not as good, proportionally, as on the X-MP series. This difference should not be overemphasized, however: the CRAY-2 still reaches half its asymptotic performance with the multiplication of matrices of order 25×25. In this sense all the Cray computers can be regarded as generally similar in achieving excellent performance without the need to go to long vector lengths. They are thus quite different from supercomputers like the CDC CYBER 205, or many of those from Japanese manufacturers.

Obtaining high performance from Cray computers requires the proper exploitation of parallelism in codes to use multiple functional units, pipelined vector hardware, and (where available) multiple CPUs. We can consider parallelism as arising at several levels. At the lowest level is the possibility of parallel execution of instructions derived from a single program statement, using the multiple functional units. At the next level is vectorization, the "parallel" execution of loop iterations using the vector functional units. From the programming point of view the fact that this is not strictly parallel but pipelined execution is usually irrelevant. It is also possible to execute different loop iterations (or groups of iterations) on different CPUs. This next level of parallelism, termed "microtasking" by CRI,[1,2] is more elaborate than the lower levels, as code must be included to acquire, release, and possibly synchronize other CPUs. The newest FORTRAN compiler release can arrange this automatically, referred to by CRI as "autotasking." An even higher level of parallelism is the execution

of larger code fragments (individual subroutines, say) on several CPUs simultaneously. This approach, referred to as "macrotasking" by CRI,[1,2] may involve execution of quite different tasks on different CPUs, whereas microtasking would usually have all CPUs executing the same instructions but with different data. Finally, the highest level of parallelism corresponds to running separate user jobs on the various CPUs: this is conventional multiprocessing at the operating system level. We shall be particularly concerned with the implementation of parallelism at these various levels in computational chemistry codes in this review.

Our aim here is to explain how various methods of computational chemistry can be formulated to take maximum advantage of the power of Cray supercomputers. In order to provide the necessary background for this we discuss the characteristics and performance of different Cray computers, and then we consider a number of computational chemistry activities in some detail. Our emphasis will be on the utilization of Cray computer parallelism, and we assume readers are already familiar with the formalism of *ab initio* electronic structure methodology[3] or of classical and quantum mechanical scattering.[4–6] We should also note here that as we routinely perform calculations using computers other than those from CRI, we generally avoid programming techniques that are very machine specific. In particular, we eschew the use of assembly language, and we generally attempt to implement algorithms that will be reasonably efficient on other supercomputer architectures. Some examples of algorithm choice motivated in part by these portability considerations are presented in our discussion.

In the next section, we discuss the various Cray computers, both hardware and software. In Section 3 we describe the performance characteristics of the different machines for some typical code fragments, and deduce from these results the appropriate techniques for obtaining maximum performance. In Section 4 and 5 we discuss the implementation of these techniques in various steps of molecular electronic structure calculations and dynamics calculations, respectively. These sections also review the ways in which supercomputers have influenced different aspects of computational chemistry. Section 6 comprises our conclusions; we also speculate on the role forthcoming machines, such as the CRAY-3 and C-90, will play in computational chemistry.

2. Cray Computers

2.1. Hardware

We shall discuss the performance and use of the CRAY X-MP, CRAY Y-MP, and CRAY-2 computers. As stated above, these are pipelined vector

processors with multiple high-speed CPUs. Vector arithmetic is performed on operands held in eight 64-element 64-bit vector registers. Results from one floating-point unit can be passed to other floating-point units while being returned to the registers on the X-MP and Y-MP: this is referred to as "chaining." This feature is not available on the CRAY-2. Our discussion of the X-MP is concerned primarily with a 4 CPU, 8 MW X-MP/48, an older machine with a clock period of 9.5 ns as opposed to the 8.5 ns of the latest X-MPs. This X-MP/48 features hardware GATHER/SCATTER and is configured with an Input/Output Subsystem with 8 MW of buffer memory and a 128 MW solid-state storage device (SSD). The main memory is divided into 32 banks, and has a memory access time of eleven clock periods to load a single word into a CPU register. Successive banks can deliver words to the registers in successive clock periods, but each bank is busy for four clock periods after initiation of a read request. Each CPU has four channels to memory: two read, one write, and one input/output (I/O). We shall also discuss some experience with an X-MP/14se, with a single CPU (clock period 10 ns) and four MW of memory. Comparisons between the performance of the X-MP/48 and the X-MP/14se are given below.

We shall discuss the performance of two slightly different CRAY-2 computers. Both are 4 CPU machines with 256 MW of main memory and a clock period of 4.1 ns. Each CPU has a single channel to memory, and 16 kilowords (KW) of very fast "local memory" that can be used as a type of cache to enhance performance. Full control over local memory is available only to the assembly-language programmer, although high-level language compilers generate code that can make some use of this hardware, as do some library routines. The CRAY-2 main memory is divided into 128 banks, but a hardware technique ("pseudo-banking") provides emulation of 256 banks. Like the X-MP, successive banks can deliver words in successive clock periods. However, the CRAY-2 memory is, in effect, divided into quadrants, and in any particular clock period a given CPU can access only one of the quadrants. This causes significant complications in a multiuser environment, as when a user job recovers a CPU and starts to issue memory requests it may well find the requests "out of phase" with the CPU quadrant access. Even in dedicated mode it is possible for a job to be up to three clock periods out of phase with memory. The two CRAY-2s discussed differ in that the older machine has a bank busy time of 57 clock periods, while the newer machine (referred to below as CRAY-2*) has a somewhat reduced bank busy time of 45 clock periods. Clearly, there is a very significant difference between the memory performance on the CRAY-2 and on the X-MP and this difference appears in almost all performance comparisons, as we will see below. We shall also present results obtained on a CRAY Y-MP, equipped with 8 CPUs (6.3 ns

clock period), 32 MW of memory (256 banks) and a 256 MW SSD. As expected, the Y-MP behaves very like the X-MP but scaled in performance by the faster cycle time.

Various disk subsystems are available from CRI; the machines to which we have access are largely configured with DD49 disk drives, with 150 MW of storage per drive and a data transfer rate on the order of 1 MW per second (MW/s) for a single unit. On the X-MP and Y-MP I/O is performed by the Input/Output Subsystem, essentially multiple I/O processors with a large memory (8 MW on our X-MP/48, 32 MW on our X-MP/14se) buffer between CPU and disk. The transfer rate between the I/O processor and CPU is more than ten times faster than the transfer rate to disk, and substantial improvements in I/O performance can be achieved by "striping" files across multiple disks, so that successive blocks are written to different drives. These blocks can be read into the I/O processor in parallel, and then the data can be transferred to the CPU. Even better I/O performance can be obtained by using the SSD, available for the X-MP and Y-MP machines. The X-MP/48 we discuss has two channels to the SSD, and each is capable of transferring data at over 150 MW/s. In addition, the SSD has no overheads associated with head positioning and rotational delays, so that I/O can be performed with essentially no I/O wait time.

In the latest operating system versions, I/O processing on the CRAY-2 also uses disk striping, and the larger memory of the CRAY-2 allows large buffers to be used for read-ahead/write-behind on sequential system I/O. Further, by taking advantage of this large memory when programming sorting steps it will often be possible to sort in memory, instead of using direct access disk I/O.

2.2. Software

2.2.1. Operating Systems

There are three operating systems in common use for Cray computers: COS, UNICOS, and CTSS. COS is a batch-job-oriented system that runs on the X-MP (and the CRAY-1). UNICOS is an interactive system for the CRAY-1, X-MP, Y-MP, and CRAY-2, based on AT&T UNIX System V with significant extensions (including some from Berkeley UNIX). CTSS is another interactive system, based on the Livermore Time-Sharing System. We will not discuss CTSS further here, nor will we consider the Guest Operating System, which allows UNICOS to be run on some CPUs of a multiprocessor system and COS on others. The X-MP/48 results we discuss are obtained under COS, while the X-MP/14se, Y-MP, and CRAY-2 results are all obtained under UNICOS.

COS, as noted above, is a batch-job-oriented system: it has a fairly limited repertoire of job control language (JCL) commands, but these nevertheless provide essentially all the functionality required to carry out large-scale scientific computations. In addition to compilers (discussed below) and loaders, COS[7] features program maintenance utilities, permanent file management and archiving, and dynamic (transparent to the user) management of resources such as the SSD or scratch file space for local files. CRI is committed to UNICOS as its recommended operating system: CRI will continue to support COS but expects to notify customers that it will be placed in "maintenance mode." Most new sites are normally installed with UNICOS as the operating system. While this may have some advantages from the point of view of operating system maintenance [COS being written largely in Cray assembly language (CAL) and UNICOS largely in C], it is not clear how much benefit the end-user sees, especially an end-user interested mainly in large-scale scientific calculations that do not involve significant workstation or graphics support. This issue is discussed at greater length below.

UNICOS[8] features essentially a complete set of UNIX commands, including both the Bourne and C shells and the TCP/IP remote file transfer (FTP) and communications (TELNET) package. UNIX itself is rather deficient in aspects of resource management and batch job execution (no queues, no scratch files, etc.), and these deficiencies are being addressed as part of the development of UNICOS. For example, the Network Queueing System (NQS)[8] has been incorporated into UNICOS in order to provide control over job execution and queues. However, UNICOS version 4.0 (the latest release at the time of writing) does not yet provide as flexible a batch environment as COS, at least from the point of view of the production user, although future releases should improve this situation. It is also unfortunate that at the present time the X-MP and CRAY-2 run under slightly different dialects of UNIX, especially in view of the reputation of UNIX as a portable operating system. Presumably the differences will continue to be eliminated as UNICOS matures.

It may be useful here to compare UNICOS and COS as operating systems as they are seen by a user interested in large-scale scientific calculations. First, there is certainly some convenience in having interactive access to the machine from the point of view of compiling, debugging, etc., although the delay in turnaround for small jobs in going through, say, a VAX station to X-MP COS should seldom be significant. Further, as many of the UNIX commands were devised to assist in progam development there is a wider range of powerful software tools available in UNICOS than in COS. On the other hand, from the point of view of running production jobs, UNICOS (strictly, UNICOS plus NQS) in its present version (4.0) appears to be inferior to COS. The dearth of resource

management facilities in UNICOS (for example, the lack of any method for allocating and managing scratch file space beyond user goodwill in CRAY-2 UNICOS) makes the COS environment much more convenient for the production user. Further, our experiences with computational chemistry codes suggest that user job throughput *on the same hardware* can be considerably (several times) higher using COS. This is related in part to the rather poor I/O performance we have experienced under X-MP UNICOS, where sequential I/O performance has been severely impacted when several jobs are running.[9] This affects CPU times as well as wall clock times, by a factor of 4 or more: we have observed a factor of 10 increase relative to COS when I/O is done with the default file block size. Finally, at this time (using COS version 1.16) none of our production tasks reveal operating system bugs in COS that require workarounds or special tactics (the issue of compiler bugs is treated below), while even under UNICOS 4.0 certain codes will not operate correctly without I/O modifications. Again, this situation will no doubt improve as UNICOS matures, but at present our view is that computational chemists are better served by COS than by UNICOS.

2.2.2. *FORTRAN Compilers*

As discussed in the Introduction, our aim is to avoid the use of assembly language and to use a high-level language for our codes. The debate over FORTRAN's deficiencies continues, but for most scientists the accumulation over a number of years of a considerable body of functioning software written in FORTRAN, together with the availability of optimizing compilers, makes it the high-level language of choice (see, e.g., Ref. 10).

The X-MP and the CRAY-2 each offer two compilers invoked as CFT and CFT77. It is a little unfortunate in view of the nomenclature that the CFT compilers on the two machines are not the same: CRAY X-MP CFT[11] is a product line that goes back to the CRAY-1, and has been fully compatible with the FORTRAN 77 standard since 1981 (version 1.10); the CRAY-2 compiler,[12] whose product name is CFT2, is a FORTRAN 66-based compiler (again originally derived from an early version of the CRAY-1 compiler) that is now compatible with the FORTRAN 77 standard. Both of these compilers are written in assembly language. CFT77[13] is (in principle) the same product on the X-MP and the CRAY-2, and as its name suggests is fully compatible with the FORTRAN 77 standard. This compiler is written in PASCAL, and as a consequence is much slower in compiling code than either version of CFT (sometimes by an order of magnitude or more), but this is only of consequence when large programs have to be completely recompiled. CRI's long-term commitment seems to be to CFT77: CFT cannot address more than 16 MW on the Y-MP, for example.

Both the CFT and CFT77 compilers feature extensive optimization and vectorization capabilities, as would be expected; CFT77 also includes explicit array syntax following the FORTRAN 8X proposal.[10] In addition, the newest releases of CFT77 provide some capability for recognizing microtasking directives[1, 2] and for automatic generation of code to run on multiple CPUs ("autotasking"). The compiler optimization involves both local and global (critical path) techniques similar in their effect to other advanced FORTRAN compilers. The compilers' ability to vectorize code has improved substantially over the years, especially in the area of conditional statements in loops, or generation of code for vectorized and nonvectorized versions of loops with run-time decision as to which is correct to use. While a number of constructs still inhibit vectorization, the compilers are being improved constantly in this area, and several examples are discussed later. As is common with vectorizing compilers, it may be necessary to inform the compiler explicitly that certain constructions should not inhibit vectorization, because from the code alone it may be impossible to tell whether vectorization should be allowed. In the loop

```
      DO 10 I=1,N
         A(I+M)=A(I)
10    CONTINUE
```

the loop can be vectorized if $M \geqslant N$, but if the compiler cannot check this at compile time, the loop would be flagged as nonvectorizable. By including a directive informing the compiler that $M \geqslant N$ always holds, that is, to ignore any potential addressing problems—"vector dependencies"—the user can ensure the loop will be vectorized. In principle, a test for such dependencies could also be generated for execution at run time, but this facility is not available at the time of writing. The compiler directives[11–13] have the form of a FORTRAN comment statement, so this does not interfere with portability; however, while other manufacturers also employ such devices there is no standardization of directive names or syntax.

For efficient use of Cray computers, it is imperative to make maximum use of data once it has been loaded into the vector registers. The compilers are still not sophisticated enough to relieve the programmer of this job entirely: for example, the unrolling of DO-loops[14] usually enhances code performance under CFT or CFT77, unlike, say, the Alliant FX/FORTRAN compiler, for which user unrolling of loops generally *reduces* performance.

The latest improvements in CFT77 notwithstanding, FORTRAN support for using multiple CPUs on the various Cray machines is still fairly primitive.[1, 2] Microtasking requires special constructs to be inserted in the code, which is then passed through a preprocessor to generate small CAL routines to manage the multitasking. It is this preprocessing step that the

latest version of CFT77 appears to be able to manage for itself. Autotasking will obviate the need for any user intervention of this sort. Macrotasking, which normally involves coarse-grained parallelism, is handled through a set of FORTRAN-callable routines to organize starting multiple tasks and synchronizing them. In the very earliest form of microtasking, it was intended that microtasked versions of various library routines would be available, but this is not the case at the time of writing. This is a great pity, as a method of using multiple CPUs that was essentially transparent to the user (in particular, that required no nonstandard code modifications) would be very valuable, as would automatic compiler-generated parallel code for DO loops (especially outer DO loops where inner DO loops have been vectorized). These facilities are already present in compilers and libraries for other machines, like the Alliant FX/8,[15,16] and are appearing in the CFT77 compiler as it is further developed. Microtasked libraries are becoming available from CRI. In view of the trend to larger numbers of CPUs (8 on the Y-MP, 16 on the CRAY-3 and C-90) it seems imperative to provide strong support for multitasking—as the number of CPUs grows it will become increasingly difficult to keep a machine busy with individual jobs unless very sophisticated job schedulers are developed.

One of the inevitable consequences of exploiting machine parallelism is an increase in the memory required for a given computational task. One of the "8X" extensions to FORTRAN is the concept of allocatable arrays,[10] storage for which is allocated dynamically on entry to a subroutine and which is deassigned (i.e., deallocated, at least in principle) on exit from the subroutine. This facility is available in recent versions of CFT77, but unfortunately the allocated storage is not returned on exit from the subroutine, so that the overall memory length can grow but not shrink using this approach. The use of allocatable arrays also incurs substantial overheads at present. More traditional methods of "dynamical allocation" are based on user-controlled expansion of the field length to adjust the size of blank common, which is conventionally loaded at the end of user memory. In this way memory can be acquired and returned as desired. However, this approach reflects a fairly primitive philosophy—that of using a user stack and assuming that no other part of the job will alter the field length. This assumption is vitiated at the outset in COS, for example, where system buffers are allocated at the highest user addresses, "growing" downwards as more files are opened. Care is therefore required to ensure that the user's expanding stack does not collide with the system. In general, with more sophisticated memory management features becoming available (like the heap-based allocation in UNICOS[17]) it seems preferable to avoid expanding blank common and to make more use of allocatable arrays. This is especially true in multitasked jobs, where special attention must be paid to avoiding conflicts in addressing data structures in different tasks. The

use of allocatable arrays within each task is clearly a much simpler approach than trying to coordinate the expansion of blank common by more than one task. It is to be hoped that this very powerful feature will be fully implemented soon, as it will provide an important part of the support for multitasking.

In view of the sophistication required of a compiler that is to perform global optimization, vectorization, and some exploitation of multitasking, it is not entirely unexpected that the Cray compilers occasionally produce erroneous code. Much of this (especially under CFT2 on the CRAY-2) derives from attempts to optimize too large and complicated a code segment, and can be cured simply by decreasing the maximum size of a code block the compiler will try to optimize. Further, errors seldom cause a code to produce answers close to the correct values—they usually produce obviously incorrect results. Nonetheless, such errors are a considerable nuisance, as they can often be data dependent, and will disappear in small, manageable test calculations, occurring only in large, expensive production runs. Unlike the situation with the UNICOS operating system, where there has been an overall improvement as new system versions have been released, we have encountered more problems with the newer compilers than with the older versions. This is particularly true of the X-MP CFT compiler. No doubt the adoption of CFT77 as the sole FORTRAN compiler will allow more effort to be concentrated on improved and error-free generation of code to exploit the various levels of parallelism.

3. Performance

3.1. Elementary Computational Chemistry Kernels

A number of previous studies of Cray computer performance in the context of computational chemistry have appeared, and timing information on various types of constructs (primitive operations or "kernels," as they are termed in the study by Saunders and Guest[18]) that occur widely in computational chemistry codes have been given. In the present work we will use two typical kernels to illustrate particular performance aspects of different Cray computers. We will then briefly present performance figures of some other representative kernels. We should note here that the timing results we present are subject to some uncertainties, especially on the CRAY-2 where memory access delays from other user jobs can make observed times reproducible to no better than 20%. Further, some of our Y-MP tests were run on a machine with a 6 ns clock period, so that performance figures for a given kernel in one table can differ by 5% from figures in another table.

The first kernel we will consider is the addition of a multiple of one vector to another, given in FORTRAN as

```
      DO 10 I=1,N
         A(I)=A(I)+SCALAR*B(I)
10    CONTINUE
```

(referred to as SAXPY, using the BLAS name[19]). This is a rather typical simple vector operation and its behavior will serve as a model for other loops. The second kernel is the more elaborate operation of matrix multiplication, represented in FORTRAN (with a SAXPY inner loop) by the code

```
      DO 10 J=1,N
         DO 20 I=1,N
            C(I,J)=A(I,1)*B(1,J)
20       CONTINUE
         DO 30 K=2,N
            DO 40 I=1,N
               C(I,J)=C(I,J)+A(I,K)*B(K,J)
40          CONTINUE
30       CONTINUE
10    CONTINUE
```

Matrix multiplication is a particularly efficient operation on all Cray computers and it therefore offers the greatest scope for enhancing program performance. In practice, what is often desired is the more elaborate expression

$$\mathbf{C} = \alpha\mathbf{C} + \beta\mathbf{AB} \tag{1}$$

which forms the proposed Level 3 BLAS specification.[20] We will discuss later the handling of the more general forms and of sparseness in the matrices.

For some simple kernels, it is possible to estimate the expected performance by considering instruction times. However, in most cases this is not very fruitful, as both compiled code and library code may take advantage of special features (or may be handicapped by special problems such as bank conflicts) and often produce quite different rates. We have chosen here to discuss only observed performance obtained on various machines (running a normal job mix, not in stand-alone mode) under different compilers. By analogy with the approach to hardware characterization of Hockney and Jesshope,[21] we can evaluate from the performance data two quantities characteristic of each kernel: the maximum performance, r_∞, (a rate in MFLOPS) and the vector length at which half this performance is

observed, denoted $n_{1/2}$. (We should note that since most operations do not yield monotonically increasing performance rates, r_∞ is not taken as the asymptotic rate, but rather as the highest observed rate for any vector length up to 1024.) In addition to the rather detailed discussions of SAXPY and matrix multiplication performance, we tabulate r_∞ and $n_{1/2}$ values for a number of other kernels concerned with both arithmetic and data motion.

3.2. SAXPY and Matrix Multiplication

Table 1 shows the performance for the SAXPY operation obtained on a CRAY X-MP/48, using the CFT and CFT77 compilers, and also the SAXPY subroutine from CRI's scientific subroutine library, SCILIB.[17, 22] It is evident that the best performance is obtained using the CFT77 compiler, avoiding the overhead associated with calling the library routine. CFT77 also displays the smallest $n_{1/2}$ value. The better performance of the code obtained from CFT77 relative to CFT is rather typical of the behavior of the two compilers (at least at the time of writing): CFT77 commonly produces code faster by 20%–30% or more. All times in Table 1 show a steady growth in performance as the vector length increases, up to a vector length of 63. Somewhat unexpectedly, in view of the "natural" vector length of the machine, the performance for vector lengths of 64 is not quite as high as for 63, although this may reflect the fact that the code generated is designed to cope with vectors longer than 64. This behavior is less

Table 1. SAXPY Performance (in MFLOPS) on CRAY X-MP/48

N^a	CFT	CFT77	SCILIB
4	5.6	11.4	4.8
8	10.9	22.7	9.4
12	16.1	34.1	13.8
16	21.1	44.7	18.6
25	30.4	68.0	26.9
32	35.6	86.3	31.6
63	59.7	132.8	56.9
64	58.3	128.6	54.5
127	71.6	119.3	69.1
128	80.8	135.5	72.7
255	95.6	131.0	100.7
256	108.0	121.2	105.2
r_∞ ($n_{1/2}$)	110 (56)	159 (30)	118 (78)

[a] Vector length.

apparent for other pairs of lengths such as 127 and 128, or 255 and 256. Finally, we note that the best rate observed for the SAXPY operation falls short of the ultimate performance (210 MFLOPS for an X-MP with 9.5 ns clock), even though both an addition and multiplication are required in the innermost loop and these operations can be chained on the X-MP.

A comparison of SAXPY performance on several different Cray computers is given in Table 2. Only the approach that gives the highest r_∞ is shown: this is the simple FORTRAN loop of the previous subsection compiled with CFT77 on the X-MP machines and the SCILIB version of the BLAS call on the CRAY-2 machines. This difference probably reflects the absence of hardware chaining on the CRAY-2, which could be partially compensated for by careful assembly language coding of a library routine but which is probably beyond the compiler's capabilities. The best performance on the CRAY-2 is much less than the theoretical 488 MFLOPS, so even less of the machine's power can be exploited this way than on the X-MP. Also, while the CRAY-2 performance improves steadily up to vector lengths of about 64 there are numerous fluctuations above this value that do not correlate with any obvious hardware characteristics. The timing tests were performed on production machines in multiuser mode, so these fluctuations may represent problems with bank conflicts engendered by other user jobs and by context switching.

Table 2. SAXPY Performance (in MFLOPS) on Cray Computers

N[a]	X-MP/14se[b]	X-MP/48[b]	Y-MP/832[b]	CRAY-2*[c]	CRAY-2[c]
4	11.1	11.4	15.2	3.7	3.3
8	22.1	22.7	30.3	7.0	6.5
12	33.2	34.1	45.5	10.1	9.5
16	44.2	44.7	60.6	13.0	12.1
25	69.0	68.0	94.5	19.1	18.8
32	84.8	86.3	120.9	23.2	22.7
63	112.9	132.8	211.5	40.3	38.1
64	103.3	128.6	213.5	42.6	37.6
127	104.4	119.3	213.1	41.1	53.4
128	103.9	135.5	214.1	52.2	50.8
255	119.6	131.0	242.1	58.0	67.8
256	104.0	121.2	242.5	80.5	62.2
300	120.5	158.6	178.7	70.8	66.2
511	121.7	143.3	206.8	63.0	45.1
512	120.3	140.6	206.5	67.2	63.1
r_∞ ($n_{1/2}$)	122 (22)	159 (30)	243 (32)	81 (63)	68 (55)

[a] Vector length.
[b] CFT77 gives best performance.
[c] SCILIB routine gives best performance.

The X-MP/48 matrix multiplication results displayed in Table 3 provide a different perspective on performance. Here, the SCILIB routine MXM outperforms all the FORTRAN implementations listed. The latter comprise the SAXPY inner loop form given explicitly in the previous subsection, a similar approach with a dot product inner loop, and the SAXPY inner loop form as above with the J loop unrolled four times. Although the unrolled loop structure gives quite good performance with larger array dimensions, it is still not competitive with the library MXM. The latter not only has an r_∞ value of almost 200 MFLOPS, but obtains half this rate with a matrix dimension of only 11. The use of the library matrix multiplication routine would thus seem to be an ideal route to high performance on the CRAY X-MP, even for problems of rather small dimension.

A comparison of matrix multiplication performance on several Cray computers is given in Table 4. For all machines the highest rates are obtained with the library MXM routine. While the CRAY-2 performance still falls somewhat short of the maximum possible, experience has shown

Table 3. Matrix Multiplication Performance (in MFLOPS) on CRAY X-MP/48

N^a	DOT[b]	SAXPY[c]	4*unrolled[d]	MXM[e]
4	2.2	12.1	9.6	22.8
8	4.6	24.3	22.2	72.4
10	5.7	29.4	27.0	92.8
12	6.8	34.1	33.6	111.4
16	9.0	40.5	42.9	134.5
25	14.3	56.8	63.1	159.9
32	17.2	67.0	75.0	172.3
50	25.8	80.4	98.0	186.2
63	33.3	91.5	103.8	190.8
64	22.5	91.7	107.0	191.3
75	31.7	87.1	96.5	180.1
100	38.6	98.5	118.2	189.9
127	53.9	113.3	127.5	194.2
128	30.2	110.8	125.0	194.3
175	70.5	120.1	131.0	194.0
255	82.5	116.2	131.9	195.8
256	39.8	117.9	135.3	195.8
400	83.0	122.2	137.8	194.8
511	103.5	124.3	139.5	196.5
512	40.8	126.3	143.7	196.6
r_∞ ($n_{1/2}$)	104 (127)	126 (127)	144 (31)	197 (11)

[a] Vector length.
[b] Dot product inner loop; CFT77.
[c] SAXPY inner loop; CFT77.
[d] Unrolled to a depth of four (see text); CFT77.
[e] SCILIB routine.

Table 4. Matrix Multiplication Performance (in MFLOPS) on Cray Computers

N^a	X-MP/14se	X-MP/48	Y-MP/832	CRAY-2*	CRAY-2
4	21.8	22.8	32.1	19.7	18.5
8	68.0	72.4	101.8	63.1	58.5
10	88.1	92.8	133.2	86.4	82.2
12	105.0	111.4	159.7	104.8	98.2
16	130.8	134.5	201.7	146.5	132.1
25	153.9	159.9	239.3	203.6	178.1
32	164.3	172.3	256.8	249.4	223.4
50	177.1	186.2	277.6	312.8	301.3
63	181.8	190.8	285.3	339.6	330.6
64	181.7	191.3	285.8	361.7	329.6
75	171.9	180.1	268.4	288.2	255.5
100	180.6	189.9	283.1	334.6	295.5
127	184.6	194.2	289.8	380.3	337.9
128	184.7	194.3	290.0	361.8	330.9
175	184.3	194.0	289.1	337.6	321.5
255	186.1	195.8	292.0	382.9	235.3
256	186.1	195.8	292.1	337.9	256.7
400	185.1	194.8	290.4	340.3	264.5
511	186.8	196.5	293.1	379.0	238.6
512	186.1	196.6	293.1	342.9	241.2
r_∞ ($n_{1/2}$)	187 (10)	197 (11)	293 (11)	383 (33)	338 (22)

[a] Vector length.

that this derives substantially from bank conflict problems created by competition with other user jobs. In stand-alone mode we have observed CRAY-2 single processor MXM performance of more than 420 MFLOPS. This is similar to the fraction of the theoretical performance obtained on the X-MP/48. Although $n_{1/2}$ for MXM on the CRAY-2 is larger than for the X-MP machines the values are still fairly small, so we can conclude that matrix multiplication is a route to high performance on the CRAY-2 as well as on the X-MP.

It is quite straightforward to rationalize the different behavior of SAXPY and matrix multiplication. In the latter, considerably more arithmetic operations can be done on data once it has been read from memory into the vector registers as compared to the former. There is therefore a much higher proportion of floating-point operations in the overall operation count in the matrix multiplication case and consequently higher performance. This observation is the key to programming Cray computers for high performance: to seek tasks that reuse data in the vector registers repeatedly, rather than using it once or twice. Matrix multiplication is probably the most obvious example of this approach, and we will generally try to show how it can be incorporated into user programs. Other related

kernels, such as solving systems of linear equations, are also considered when we discuss particular computational chemistry tasks. In view of this central importance of matrix multiplication, we shall explore some aspects of it further here.

Reference has already been made to more general matrix multiplication operations such as equation (1). Although this form is not available in the SCILIB library, not even in the simple form involving the constraints $\alpha = 1$ (or 0) and $\beta = \pm 1$, several groups have prepared subroutines to perform this task. For the CRAY-2 the subroutine MXMPMA by Calahan *et al.*[23] handles the above form with the constraints $\alpha = 1$ or 0 and $\beta = \pm 1$. This is a very efficient implementation that exploits local memory on the CRAY-2. Another aspect of more general matrix multiplication is the exploitation of sparseness in the arrays A and B. Where sparseness has its origins in the symmetry of a problem, it is usually more advantageous to handle the symmetry explicitly, but it may often be useful to take advantage of essentially random sparseness. Saunders' routine MXMB[18] is widely used in this situation: its performance is similar to the CRI MXM routine for dense matrices so there is no loss of efficiency where there is little sparseness.

Another generalization of the simple matrix product is the case in which either A or B is to be transposed before the multiplication. The SCILIB routine MXMA allows for this possibility (and for other variations on the indexing of the matrices, such as multiplication of sub-blocks of full arrays). The alternative approach would be to explicitly transpose the arrays as required. On the CRAY X-MP/48 it is our observation that there is little to choose between these strategies. On the CRAY-2 for most array dimensions there is also little difference, but there is a catastrophic loss of performance in MXMA when transposing arrays with dimensions that are multiples of 256 because of severe bank conflicts. In such cases the observed performance drops to some 18 MFLOPS, about 20 times slower than the best rate. Several smaller dimensions, such as 128 or 64 also show a drop in performance, although not as great. While this can be overcome by embedding the desired matrix in a larger array to avoid the critical stride value, we normally prefer to explicitly transpose the matrix and use the routine MXM. Since, as noted, this strategy incurs virtually no penalty on the X-MP, we follow it on all Cray computers. This also increases the portability of programs, as it usually easier to find an analog of MXM on other machines than MXMA.

The technique of obtaining increased performance on Cray computers by unrolling an outer loop is well documented,[14] but it is perhaps less obvious that this approach can be extended to more than one loop. In the matrix multiplication case this gives the following code fragment when two loops are unrolled to a depth of two:

```
      DO 10 J=1,N,2
         DO 20 K=1,N,2
            DO 30 I=1,N
               C(I,J)=C(I,J)+A(I,K)*B(K,J)
                           +A(I,K+1)*B(K+1,J)
               C(I,J+1)=C(I,J+1)+A(I,K)*B(K,J+1)
                           +A(I,K+1)*B(K+1,J+1)
30          CONTINUE
20       CONTINUE
10    CONTINUE
```

Unrolling the J loop enhances performance by reducing the number of fetch instructions required for the elements of A, while unrolling the K loop reduces the number of fetch and store instructions for the elements of C. As A and C are different arrays, these advantages are combined when both loops are unrolled, and the overall efficiency is increased. Formally, the efficiency increases as the unrolling depth increases, but the finite number of vector registers, together with the limits on the size of code block the compilers can optimize imposes an upper limit to the useful depth of eight. CRAY-2 timings obtained with such a scheme (both loops unrolled eight times) are given in Table 5. Although the performance is considerably improved over the case of FORTRAN loops without unrolling, it is never as

Table 5. Performance of Specialized Matrix Multiplication Techniques (in MFLOPS)

	CRAY X-MP/48		CRAY-2*			
N^a	MXM[b]	8*unrolled[c]	MXM[b]	8*unrolled[c]	Strassen[d]	MXMPMA[e]
16	136	52	111	56	93	110
63	191	95	319	110	364	370
64	191	103	285	168	367	373
127	194	111	289	187	385	386
128	194	112	381	225	383	378
255	196	112	259	188	386	387
256	196	113	304	233	385	383
511	197	117	387	242	407	387
512	197	117	328	259	423	387

[a] Vector length.
[b] SCILIB routine.
[c] Unrolled to a depth of eight (see text); CFT2.
[d] Strassen algorithm (Bailey, Ref. 27).
[e] Calahan *et al.*, Ref. 23.

good as with MXM, so the latter would usually be preferred. An exception might be the case of equation (1) with α different from zero. The unrolled scheme can handle this case without the need to store a temporary product matrix separately, as would be required if MXM was used.

For very large matrices there is some advantage in using specialized matrix multiplication algorithms that require fewer than $2n^3$ floating-point operations.[24–26] Many such algorithms use a recursive partitioning approach akin to Fast Fourier Transforms, in which the matrices are multipled by blocks using expression rearrangement to eliminate some operations. The best of these schemes behaves as roughly $n^{2.5}$,[26] but a simpler scheme, due to Strassen,[24] which behaves as about $n^{2.8}$, has recently been investigated on the CRAY-2 by Bailey.[27] We can compare this approach with the library routines and also with loop unrolling in FORTRAN. The results are displayed in Table 5. The routine MXMPMA is the assembly language program written by Calahan *et al.* referred to above.[23] It is this routine that is used to perform the block matrix multiplications in the Strassen scheme. The performance figures for the latter are computed assuming $2n^3$ floating-point operations: for the large matrices (say, larger than 500×500) they are distinctly better than the other values. Note also that MXMPMA does not suffer from the bank conflicts that somewhat degrade the library MXM performance for the 256×256 and 512×512 cases.

3.3. General Performance

Observed performance, given as r_∞ and $n_{1/2}$ values, for a number of operations are given in Table 6. The SAXPY and matrix multiplication results of the previous subsection are also reproduced for reference. For elementary vector operations like addition, the code produced by the CFT77 compiler out-performs the library subroutines on the X-MP and Y-MP, but the library versions run faster on the CRAY-2. Performance figures are also given for sparse vector operations "SPDOT" and "SPAXPY".[22] These allow for sparseness to be exploited in one vector: the SPAXPY for example is equivalent to the FORTRAN loop

```
      DO 10 I=1,N
         A(INDEX(I))=A(INDEX(I))+SCALAR*B(I)
10    CONTINUE
```

and is useful when vector **B** has been compressed down to nonzero values whose original addresses are stored in INDEX. SPDOT is simply the dot product equivalent. These kernels are useful in quantum chemical calculations: the performance of the library versions is quite good.

Table 6. Performance (in MFLOPs) for Different Operations[a]

Operation	X-MP/14se	X-MP/48	Y-MP/832	CRAY-2	CRAY-2*
Vector add	60 (24)	70 (24)	119 (28)	39 (63) L	48 (72) L
Dot product	90 (96)	95 (91)	148 (102)	80 (132) L	103 (96) L
SAXPY	122 (22)	159 (30)	254 (31)	68 (55) L	101 (128) L
Vector divide[b]	25 (16)	26 (15)	41 (16)	24 (23)	26 (26)
MXV[c]	187 (17) L	195 (18) L	312 (19) L	242 (27) L	294 (27) L
MXM[d]	187 (10) L	197 (11) L	308 (12) L	338 (22) L	383 (33) L
Cubic[e]	168 (17)	174 (15)	288 (19)	113 (15)	121 (17)
Sextic[e]	178 (14)	187 (13)	297 (16)	142 (17)	148 (17)
[2/1][f]	104 (13)	109 (12)	170 (13)	116 (16)	118 (17)
[3/2][f]	125 (11)	132 (11)	207 (12)	154 (16)	158 (17)
square root[b]	11 (15)	12 (16)	14 (15)	23 (21)	24 (19)
sine[b]	3 (9)	4 (10)	5 (11)	6 (13)	6 (16)
arctangent[b]	5 (11)	5 (11)	8 (13)	6 (16)	6 (15)
exponential[b]	6 (12)	6 (12)	9 (13)	9 (25)	9 (25)
$\log_e$[b]	4 (11)	4 (11)	6 (11)	6 (16)	6 (15)
MOVE[g]	76 (23)	83 (25)	143 (30)	51 (37)	72 (47)
GATHER[h]	26 (10)	46 (14)	80 (19)	17 (22)	20 (25)
SCATTER[h]	32 (11)	45 (13)	88 (20)	20 (16)	24 (16)
SPDOT[i]	32 (67) L	34 (64) L	52 (92) L	40 (230) L	48 (192) L
SPAXPY[i]	44 (58) L	47 (55) L	77 (66) L	31 (63) L	42 (96) L

[a] L denotes SCILIB routine; otherwise CFT77. $n_{1/2}$ values in parentheses.
[b] Performance in megaresults per second.
[c] Matrix–vector product.
[d] Matrix multiplication.
[e] Polynomial of given order with vector of arguments.
[f] Rational fraction of given order with vector of arguments.
[g] Vector move, performance in MW/s.
[h] Performance in MW/s.
[i] Sparse vector operations (see text).

Polynomial evaluation using Horner's rule, such as

```
      DO 10 I=1,N
         A(I)=((C3*X(I)+C2)*X(I)+C1)*X(I)+C0
10    CONTINUE
```

for the cubic case, is very efficiently vectorized by CFT77 on the X-MP and Y-MP, where for higher polynomial orders a substantial fraction (about 90%) of the maximum machine performance is obtained. This indicates that the compiler is able to generate code to reuse data from the vector registers for this case. Polynomial evaluation is a less fruitful approach on the CRAY-2, where the convergence with increasing polynomial order is much slower.

Element-by-element vector division ("vector divide") is also shown in Table 6. Cray computers have a reciprocal approximation unit but no

floating-point divide hardware, so division requires a reciprocal approximation, a multiplication to obtain an estimate of the quotient, followed by a Newton iteration to give the required accuracy in the quotient (see, e.g., Ref. 21). This step requires a subtraction and two more multiplications. Even if the subtraction is chained with one multiplication this process cannot run at more than 40 MFLOPS (strictly speaking at 40 megaresults per second) on our X-MP/48, and the observed performance is noticeably less. However, if the division is part of a more elaborate computation, for which data can be held in the vector registers and re-used, the overall performance will be more satisfactory. The [2/1] rational fraction evaluation requires the same number of simple additions and multiplications as the cubic polynomial, and also requires division, with its contributing multiplications and subtractions, but the observed performance is still about two-thirds of the polynomial rate on the X-MP and Y-MP, and is equal to the polynomial rate on the CRAY-2.

For many purposes it is necessary to evaluate special functions, and the performance of the FORTRAN library routines for several of these is shown in Table 6. The rates here are quoted in megaresults per second, and as these functions can require 20 or more floating-point operations for their evaluation the performance is seen to be rather good. It is interesting that this is one of the few areas in which the CRAY-2 outperforms the X-MP. On all machines the square root performance is substantially better than the other functions but is still not competitive with elementary vector operations.

The last type of operation considered in Table 6 is data motion. On the CRAY-1 data motion was an expensive task, with a performance for a simple vector move of about 8 MW/s. With improvements such as hardware GATHER/SCATTER all of the current Cray computers perform much better than this, with vector move rates that are higher than operations like addition. Data motion on the CRAY-2 shows a somewhat worse performance ratio to the X-MP than that seen for elementary vector operations.

3.4. Multitasking on Cray Computers

As discussed in the Introduction, vectorization is not the only way to improve performance on those Cray computers with several CPUs: it is also possible to use several CPUs in parallel. The pros and cons of this approach, especially in the context of multiuser systems, have been discussed elsewhere.[28] Multitasking involves some overheads, so the *total* CPU time of a job will *increase* when it is multitasked.[1, 2] Any timing improvement therefore comes (in a stand-alone machine) from a reduction in wall clock time. Except for the case of large user jobs that would be

feasible only if run multitasked, where there is obviously no alternative, system throughput on a production machine is generally best served by minimizing user multitasking and maximizing multiprocessing (that is, "system multitasking"). There can be circumstances that demand a different approach, however. For example, we have previously discussed the "$1/n$ rule":[28] on a system with n CPUs there will be no risk of CPUs becoming idle because of lack of system resources provided users are allowed to acquire no more than $1/n$ of any resource. Clearly, if a user job *requires*, say, all of main memory the user should be encouraged to use all the CPUs, in order to keep the entire machine busy. This is thus another motivation for investigating multitasking.

We concentrate here only on the performance obtained with multitasked codes. First, it should be noted that in any multiprogrammed system the opportunity to use multiple CPUs will vary considerably with workload and the various tuning parameters in the job scheduler. As a consequence, the performance measured for a multitasked job will commonly vary by a considerable amount from run to run, making it difficult to compare different approaches and different machines. Some of the results presented here were therefore obtained during dedicated access to a given machine. Second, multitasking is not yet a commonplace activity on Cray computers, at least in our observation, and therefore some aspects of multitasking have yet to be properly debugged. Finally, we should point out that, as a corollary of using dedicated time, some performance figures for the CRAY-2 are quite different from those given in the previous section. The reasons are discussed below.

Table 7 contains performance figures for matrix multiplication using different numbers of CPUs on different machines. This is a macrotasked matrix multiplication in which the result matrix is divided into n column groups (for n tasks), as described in detail in Ref. 28. The X-MP and Y-MP display an essentially linear scaling with the number of CPUs employed, producing the impressive figure of more than 2.3 GFLOPS when eight CPUs are used on the Y-MP. The overheads associated with generating the tasks and acquiring CPUs are not included in Table 7, but if the matrix

Table 7. Macrotasked Matrix Multiplication Performance (in MFLOPS)[a]

	1 CPU	2 CPUs	4 CPUs	8 CPUs
X-MP/48	200	399	796	
CRAY-2	420	756	1216	
Y-MP/832	293	585	1166	2320

[a] Obtained in stand-alone mode on X-MP/48 and CRAY-2. In stand-alone mode the Y-MP rates would be some 3%–4% higher.

multiplication is repeated several times within the job, as would probably be the case in a production code, the mean overhead becomes negligible anyway. The CRAY-2 figures scale considerably worse than n, even in a stand-alone environment. This has been discussed at length elsewhere: it stems from bank conflicts in one task generated by the other tasks. In a production environment the performance is even worse, as more bank conflicts arise from other user jobs and from the context switching and job swapping that occurs in multiuser mode. It is this interference between tasks that makes such a difference between stand-alone and multiuser mode times for even single-tasked jobs on the CRAY-2; it is noteworthy that our stand-alone multitasked matrix multiplication performance results on four CPUs are about four times the multiuser mode single-tasked results of Section 3.2.

There is little value in repeating the above figures for microtasked versions of the matrix multiplication. These show essentially the same performance improvements as does macrotasking in a stand-alone environment, while the improvement possible in a production machine is an even stronger function of the workload than is the case for macrotasked jobs, so there is no simple way to quantify the performance observed. Microtasking is designed to utilize idle CPUs, and under UNICOS this appears to be implemented by giving extra tasks forked a very low priority (a high "nice" value, in UNIX terminology[29]). This means that these forked tasks will get a very small share (although not a zero share) of the CPUs, unless the number of user jobs in the system falls to such a level that only the forking task and its children remain, in which case all will utilize the CPUs. Obviously, in a dedicated environment a single microtasked job will utilize all CPUs fully and perform like its macrotasked equivalent, while in a normal production environment the forked tasks will accumulate almost no time and the result will be equivalent to the single-threaded version. On the other hand, an "idle CPU" under COS seems to correspond to a CPU not busy with a task with the same priority as the spawning task. Hence a high-priority job under COS will generally be able to acquire CPUs as it desires, greatly improving its throughput (and degrading that of other user jobs), while a similar job run at lower priority will see much less improvement (if any). COS microtasking can thus impact other users more than UNICOS microtasking, and for many COS production environments microtasking may be discouraged.

Overall, while multitasking can often improve performance, its use can be counterproductive in a multiuser environment. Under COS, at least, it appears desirable to have about eight jobs per CPU in the machine to ensure that no idle time accumulates, and at present (with Cray computers of up to four CPUs capable of running COS) it should be possible in most production environments to keep a machine busy simply by multi-

processing user jobs. This might not be the case with 16 or even more CPUs, as the sophistication required in a job scheduler to cope with keeping so many CPUs busy would be considerable. It may also be the case that for some environments in which very long jobs or jobs with very heavy resource requirements are run that it is not possible to schedule jobs to keep all CPUs busy. Such situations are probably better suited to the use of multitasking, at least for some part of the time. A useful compromise for many users would be to have microtasked versions of various library routines (especially matrix multiplication) available, at least under UNICOS where there is less impact of microtasking on other users. In this way the multitasking would be transparent to the user, but any idle CPUs could be utilized effectively. As noted above microtasked libraries are becoming available from CRI.

4. *Ab Initio* Quantum Chemistry

4.1. General Observations

Many aspects of *ab initio* quantum chemistry program implementations for Cray computers have been discussed elsewhere.[18, 30] We concentrate here on illustrating the general principles outlined above for efficient use of Cray computers, with examples from the MOLECULE[31, 32] and SWEDEN[33] program systems. We discuss the evaluation and sorting of integrals, the optimization of SCF, MCSCF, and CI wave functions, and also several less conventional methods for different types of electronic structure calculations. As several authors have emphasized,[34–36] the calculation of energy derivatives (gradients, Hessians, etc.) shares many features with the calculation of the energy itself, and for exigencies of space we shall only discuss the latter here. Most of the techniques described can readily be implemented in derivative calculations as well.

There is no intention here to instruct the reader in the details of efficient programming; the "Optimization Guide" published by CRI for the X-MP series[37] can be consulted for such material. We shall discuss some of the broader aspects of programming Cray computers, again under the assumption that the codes to be written must be fairly easily ported to other environments.

4.2. Gaussian Integrals and Integral Sorting

Several reviews of methods for evaluating Gaussian integrals, with special emphasis on vectorization, are available[38–40]: the comprehensive

and lucid article by Saunders[38] is particularly recommended. The evaluation of one-electron integrals is straightforward and requires little time, and we deal only with two-electron integrals here. We shall concentrate here on special features of the MOLECULE integral program that are not covered by these reviews. MOLECULE evaluates integrals over symmetry-adapted linear combinations of contracted Gaussian functions; these contractions can be of either general[41] or segmented[42] type. Basis functions can be either Cartesian Gaussians or spherical harmonic functions—in the latter case it is possible to include not only "pure" spherical harmonics (1*s*, 2*p*, 3*d*, 4*f*) but also the higher principal quantum number "contaminant" functions 3*s*, 4*p*, etc. As the previous reviews of integral generation have only sketchily treated general contractions, symmetry adaptation and transformation of Cartesians to a spherical harmonic basis (note that Saunders discusses in detail the calculation of integrals directly in a spherical harmonic basis), we shall treat these matters in some detail here. General contraction and transformation to spherical harmonics are both rather time-consuming, but the major part of the overall time is usually spent evaluating primitive integrals.

The algorithm actually used for evaluation of primitive two-electron integrals in MOLECULE is based on the traditional factorization[43] into a product of terms each involving one Cartesian direction, and an "auxiliary function" term that must be evaluated by numerical approximation, symbolically

$$I = \sum_m X_m Y_m Z_m F_m(t) \tag{2}$$

where m runs from zero to a limit given by the sum of angular quantum numbers involved; $F_m(t)$ is the "auxiliary function," t depends on the relative distance between the various basis functions and their exponents. The efficient implementation of this approach requires that all functions from a given shell (that is, that differ only in azimuthal quantum numbers) be treated together. The factors X_m can be obtained by recursion, as shown originally by Boys[44] and as exploited in various forms by McMurchie and Davidson[45] and, recently, by Obara and Saika.[46] The recursion scheme used in MOLECULE has many features in common with the latter approach, although it was originally derived some years ago to help implement efficiently the integral formulas given by Huzinaga and co-workers.[43] Like most integral algorithms, this lends itself to "extrinsic vectorization,"[18] that is, a suitable number of quadruplets of function exponents (in effect, a list of t values) are treated together, so the various manipulations involved in equation (2) are performed for vectors of the desired length. This approach to vectorization is simplest if the terms

grouped together are on the same set of four centers, so that when there are only one or two functions of a given type on each of the centers the vector lengths would be very short. The use of large primitive sets is becoming more common, however, and this produces satisfactory vector lengths in most cases. The values in the vector of t values vary over a wide range in most calculations, and there is considerable sparseness to be exploited in the use of equation (2). It is convenient to compress the vector to only nonzero values, compute the auxiliary function values for the compressed list and then expand the result list back for use in (2). This can be done in practice without requiring an index vector of the same length as the vector of t values because the latest version of X-MP CFT can vectorize the loop:

```
      J=0
      DO 10 I=1,N
         IF (A(I) .GT. THR) THEN
            J=J+1
            B(J)=A(I)
         ENDIF
10    CONTINUE
```

Since the vector of t values can number upwards of 50,000 terms for a large basis set, the elimination of the index vector can be very valuable on the X-MP. The EXPAND operation that is the reverse of this compression is also vectorized by X-MP CFT.

Given then a set of primitive integrals, how can these be contracted efficiently using general contractions? This is, in effect, a four-index transformation

$$(ij|kl) = \sum_p \sum_q \sum_r \sum_s (pq|rs) T_{pi} T_{qj} T_{rk} T_{sl} \tag{3}$$

but in cases where, say, six to ten primitives are contracted to two or three contracted functions, the vector lengths in (3) are simply too short for effective performance when formulated in the conventional matrix multiplication scheme (discussed in Section 4.4 below). It is preferable to use a different approach for each of the four partial sums in (3): if the $(pq|rs)$ are arranged in a rectangular matrix I with column index p and a "compound" row index qrs, the first partial sum for (3) can be written

$$I'_{qrs,i} = \sum_p I_{qrs,p} T_{pi} \tag{4}$$

For N primitive functions and n contracted functions this is, in effect, a

matrix multiplication of an $N^3 \times N$ matrix by an $N \times n$ matrix,[47] where the more conventional scheme would have *rs* fixed and would thus involve only $N \times N$ matrices. As the Cray library matrix multiplication routines are organized to perform the work in the order that gives maximum vector lengths, the scheme (4) will be vectorized with an inner loop of length N^3. The next partial sum requires a "transposition" of the result array I′ to give indices *q* and the compound *rsi*, but this can be done straightforwardly either by an explicit transposition loop or, implicitly, by using the Cray library routine MXMA (see Section 3.2 above) to multiply matrices with nonunit stride between elements. For the CRAY-2, as discussed above, the explicit transposition has the virtue of reducing the performance losses arising from bank conflicts: no extra memory is required for the transposition as the transposed I′ can overwrite the original I. The simplest solution is thus to use MXM throughout.

Once the contraction has been performed for all sets of angular quantum numbers in a batch, it is possible (if desired) to transform to a basis of spherical harmonic functions. There are several advantages to such a transformation. First, and perhaps most importantly, if only the pure spherical harmonics are used the dimension of the basis (and therefore the length of the integral file) is reduced. Even the decrease in *d* shells from six components to five can bring about a useful reduction in the length of an integral file, while for a diatomic molecule the elimination of the contaminants from a [5*s* 4*p* 3*d* 2*f* 1*g*] basis will reduce the length of the integral file by considerably more than half. Again, the size of the matrices that are involved in this process is rather small, especially for the commonest higher angular momentum cases (*d* and *f* shells), so that it is advantageous to use the same scheme as is used for the contraction transformation. However, the arrays of transformation coefficients from Cartesians to spherical harmonics are rather sparse, especially for the lower angular quantum numbers, so it is useful here to have a matrix multiplication scheme that can exploit sparseness. Nevertheless, even with the Cray library matrix multiplication routines the work associated with the transformation to spherical harmonics is a rather small fraction of the total integral time.[32] Incidentally, it may be useful to retain the contaminants (and perhaps even eliminate the high angular spherical harmonics) if desired, as methods that need continuumlike functions may require fewer functions of 8*p*-type to describe a *p*-wave than they would of the usual 2*p* functions.

While the transformation from contracted Gaussian functions to symmetry-adapted linear combinations (restricted in MOLECULE and in our discussion here to D_{2h} and its subgroups) could be regarded as a four-index transformation, various formal simplifications make an alternative approach preferable. Most symmetry-adaptation procedures[48,49] can be

viewed as implementations of the method of double coset decompositions presented by Davidson.[50] A symmetry-adapted integral is given as

$$(p_\alpha q_\beta | r_\gamma s_\delta) = N \sum_i \sum_j \sum_k \sum_l \chi^\alpha(g_i)\, \chi^\beta(g_j)\, \chi^\gamma(g_k)\, \chi^\delta(g_l)(\hat{g}_i p \hat{g}_j q | \hat{g}_k r \hat{g}_l s) \qquad (5)$$

where α... indexes irreducible representations, $\hat{g}_i$... are operators from the point group $\mathcal{G}$, p... are atomic basis functions, and N is a constant involving both normalization of the symmetry orbitals and selection rules on symmetry species. This expression can be replaced[50] by the simpler form

$$(p_\alpha q_\beta | r_\gamma s_\delta) = N' \sum_J \sum_K \sum_L \chi^\beta(g_J)\, \chi^\gamma(g_K)\, \chi^\delta(g_L)(p \hat{g}_J q | \hat{g}_K r \hat{g}_L s) \qquad (6)$$

Here N' now incorporates the rules which select the unique α given the three other irreducible representations. It should be noted first that the number of summations in (6) is three, compared to four in (5): (6) clearly involves less work. Second, the range of J, K, and L in (6) is easily restricted so that only unique atomic integrals need be computed and employed in obtaining the symmetry-adapted integrals.[48–50] Equation (6) can be vectorized very simply: as a group of atomic integrals with the same angular properties and centers are multiplied with the same factors in (6), each term in the triple summation can be viewed as an element in a SAXPY operation. Thus the "symmetry transformation" can be vectorized as a set of SAXPY operations with length n^4 (in the above notation). In fact, by forming the symmetry integrals for multiple quadruplets of irreducible representations at the same time, the atomic integral values can be reused several times once they have been read into registers, further improving the efficiency.

It can be seen from the foregoing paragraphs that the various operations we have discussed will involve either considerable data motion, or the use of nonunit strides. Thus the contraction of primitive integrals requires the processing of quadruplets of primitives for each quadruplet of angular quantum numbers, while the transformation to spherical harmonics requires the processing of quadruplets of angular quantum numbers for a given quadruplet of contracted functions. Finally, the generation of symmetry integrals requires the processing of different quadruplets of centers, for a given quadruplet of angular quantum numbers and contracted functions. Clearly, rather sophisticated index manipulations are required. Finally, we have tacitly ignored the use of permutational symmetry: in particular, when the pairs of primitives that form the sets of charge distributions are the same, numerous square blocks can be reduced to triangular arrays of the distinct elements. This also complicates the index manipulations.

CRAY X-MP/48 timing results for some integral calculations on the molecule N_2 are given in Table 8. The basis set used is a $(13s\ 8p\ 6d\ 4f\ 2g)$ primitive set contracted to $[5s\ 4p\ 3d\ 2f\ 1g]$ using an atomic natural orbital general contraction.[51] The contracted set comprises 140 Cartesian Gaussians, or 110 spherical harmonic basis functions when the contaminants are removed. We shall use this N_2 calculation for timing comparisons throughout this section. In Table 8 it can be seen that the transformation to spherical harmonics requires some 65 s on the X-MP/48, but the resulting integral file is less than one-third the length of the Cartesian case. For this N_2 calculation, the saving in time in all later steps as a result of using spherical harmonic functions totals less than 35 s, so there is a net increase of CPU time required when spherical harmonics are used. The substantial reduction in external storage required makes the trade-off worthwhile in most circumstances, however.

Table 8 also contains results for the N_2 system treated in lower than the maximum possible (in MOLECULE) D_{2h} symmetry. The C_{2v} group used here has the C_2 axis along the bond, so the two centers are treated as inequivalent. The C_s group is a subgroup of this C_{2v} group. As the symmetry is lowered, the integral time increases, although only slightly on going from D_{2h} to C_{2v}. In this case there is a compensation between a reduction in overhead (as there are no equivalent centers to be generated), and an increase in the number of integrals to be calculated.

A comparison of the performance of the integral generation code on different Cray computers is presented in Table 9. The integrals are generated in somewhat less time on the CRAY-2 than on the X-MP/48: this is rather unusual since for most steps in electronic structure calculations we usually observe the reverse. The improved performance on the CRAY-2 may derive in part from the use of somewhat more memory—5 MW on the CRAY-2 as opposed to 2 MW on the X-MP—which allows longer vectors to be used in the primitive integral calculation and reduces some overhead. The Y-MP performance for the integral generation is very good: the Y-MP version uses 3.5 MW of memory, so some extra improvement relative to the X-MP is expected beyond the shorter Y-MP clock period. What is observed is actually more than a factor of 2, considerably larger than is seen for other codes. No ready explanation for this behavior offers itself, it may result from more efficient code generation on the Y-MP.

In addition to the parallelism discussed so far, integral generation lends itself readily to coarse-grained multiprocessing.[28] As each batch of integrals is computed independently, and assuming the integrals can be produced in more or less arbitrary order (as discussed below they will generally need some reordering later anyway), it is possible to compute different batches in parallel. In fact, instead of spawning a new task for each batch it is simpler to divide the range of the fourfold loops over shells

Table 8. Gaussian Integrals and SCF Timings for N_2 on CRAY X-MP/48[a] (in seconds)

	D_{2h}^{b}		D_{2h}		C_{2v}		C_s		C_1	
	N_{INT}^{c}	Time	N_{INT}	Time	N_{INT}	Time	N_{INT}	Time	N_{INT}	Time
Integrals	6230657	407.8	1837575	472.4	3426083	477.3	8416943	955.6	15741082	1676.3
					Ordering of symmetry integrals[d]					
Symmetry	N_{ORD}^{e}	Time	N_{ORD}	Time	N_{ORD}	Time	N_{ORD}	Time	N_{ORD}	Time
$\alpha\alpha\alpha\alpha$	509026	0.5	162710	0.2	1237527	0.6	6847625	12.6[f]	37271025	62.9[f]
$\alpha\beta\alpha\beta$	3411984	7.9[f]	1223503	1.2	3718896	6.1[f]	7840000	14.9[f]		
$\alpha\alpha\beta\beta$	940545	1.7	347383	0.7	995841	1.2	2037700	5.0[f]		
$\alpha\beta\gamma\delta$	3324672	10.2[f]	1321917	2.1	1249248	1.3				
Total	8186227	20.4	3055513	4.3	7201512	9.3	16725325	32.5	37271025	62.9
					SCF calculation					
	F[g]	Iter[h]	F	Iter	F	Iter	F	Iter	F	Iter
	0.4	0.6	0.2	0.4	0.4	0.7	0.6	1.2	1.0	2.2

[a] [5*s*4*p*3*d*2*f*1*g*] basis; spherical harmonic functions used except where indicated.
[b] Cartesian basis functions.
[c] Number of nonzero integrals computed.
[d] Ordering performed in memory unless otherwise specified.
[e] Number of ordered integrals generated.
[f] Ordering uses direct access storage (SSD).
[g] Closed-shell Fock matrix construction.
[h] Closed-shell SCF iteration time.

Table 9. Integral, SCF and Transformation Timings for N_2 (in seconds)

		X-MP/48	CRAY-2*	Y-MP/832
		Integral calculation		
	N^b_{INT}			
Integrals	1837575	472.4	435.8	210.8
		Integral ordering		
Symmetry	N^c_{ORD}			
$\alpha\alpha\alpha\alpha$	162710	0.2	0.4	0.1
$\alpha\beta\alpha\beta$	1223503	1.2	2.7	0.6
$\alpha\alpha\beta\beta$	347383	0.7	1.5	0.4
$\alpha\beta\gamma\delta$	1321917	2.1	5.2	1.2
Total	3055513	4.3	9.9	2.3
		SCF calculation		
Fock matrix[d]		0.2	1.3	0.2
SCF iter[e]		0.4	1.4	0.3
		Integral transformation[f]		
Transformation		8.6	8.3	6.6

[a] [5*s*4*p*3*d*2*f*1*g*] spherical harmonic basis.
[b] Number of nonzero integrals computed in D_{2h} symmetry.
[c] Number of ordered integrals of each symmetry type. All ordering is done in memory.
[d] Closed-shell Fock matrix construction.
[e] Closed-shell SCF iteration.
[f] $1\sigma_g$ and $1\sigma_u$ MOs frozen in transformation.

in the integral code and execute each subrange as a separate task. With such large granularity the overhead associated with macrotasking becomes an insignificant fraction of the total time, and in a dedicated environment with n CPUs throughput improvements of very close to n times are seen.[28]

As will become clear in the succeeding subsections, it is highly desirable to have the integrals ordered on the integral file. This is seldom consistent with efficient integral generation, especially when symmetry is used both to reduce the number of distinct integrals and in obtaining final integrals over symmetry-adapted functions. Consequently, it is usually necessary to reorder the integral file. The reordering of the one-electron integrals is, of course, trivial and we will discuss only the two-electron case. The input/output aspects of this process, especially the use of direct access storage, have been comprehensively reviewed before[52]; we will therefore concentrate here on vectorization and discuss input/output only where it has some impact on vectorization.

The MOLECULE program generates four files of two-electron integrals, partitioned according to the symmetry block structure of the integrals. For D_{2h} and its subgroups, there are four types of allowed quadruplets of symmetry species: $\alpha\alpha\alpha\alpha$, $\alpha\beta\alpha\beta$, $\alpha\alpha\beta\beta$, and $\alpha\beta\gamma\delta$, where α, etc. denote irreducible representations. Each of these types is written to a separate file, as each type can be sorted independently of the others to give the final ordered list. Each raw integral file comprises nonzero integrals together with labels containing the irreducible representations of the four basis functions and their offsets within symmetries. The first step on reading a buffer of integrals is to unpack these labels and identify the position(s) in the final ordered list for the given integral. The label unpacking, which consists of Boolean operations, and the computation of the final position, which consists of multiplications and additions, can all be vectorized over the number of elements in the buffer. The label packing in MOLECULE ensures that the compound indices in the label are strictly nonincreasing within a label, so that no conditional structures are necessary in vectorizing the label processing during the read of the raw integral file. Although the calculation of the position addresses involves (slow) integer multiplication and division, X-MP CFT and CFT77 can generate code for performing these operations in the floating-point units (the CFT compiler requires the directive FASTMD[37]). This improves the performance of the loops involved by up to a factor of 20, and of the program as a whole by more than a factor of 2. If the number of ordered integrals of a particular type is small enough to fit in main memory (which will often be the case for Cray computer memories and high symmetries) the final position addresses simply represent SCATTER pointer elements into the ordered list, and the raw integrals can therefore be placed in the desired locations with a SCATTER operation, again with a length equal to the number of elements in the buffer.

When the number of ordered integrals of a given type is too large to fit in main memory, a bin sort of the Yoshimine type[52] is employed. The final position indices are used to identify the bins for each integral and its label. Here, because of the possibility that a bin may be full and require emptying before an integral/label pair can be written to it, we encounter a step in the reordering that cannot straightforwardly be vectorized. When these bins are read back, however, the only processing required is to use the final position index, offset by the start of the subrange of integrals in a particular chain of bins, as a SCATTER pointer for the integrals. Hence this part of the work is again vectorizable with a length equal to the number of integrals in a bin. On Cray computers this length can usually be several thousand. In fact, since the SCATTER operation will move about 40 MW/s on the X-MP or 20 MW/s on the CRAY-2, it would appear to be necessary to obtain at least the same I/O transfer rates into memory to

prevent the reordering from becoming I/O bound. With head contention and rotational delays, and disk performance on nonstriped mass storage systems, this is hardly possible if the direct access bin file is disk-resident. However, the transfer rate from an X-MP SSD into memory is around 156 MW/s per SSD channel. (Although some configurations have two SSD channels, it is extremely unlikely in practice that a single user job will simultaneously have access to more than one channel.) Thus if the bin file is entirely SSD resident the reordering step will never become I/O bound. With current SSD sizes up to 512 MW and with gigaword (GW) sizes available in the near future this will normally be the case. SSDs are not available for CRAY-2 computers, but for a 256 MW CRAY-2 it will very often be possible to reorder the integrals entirely within memory. Only for large basis sets (upwards of 200 basis functions) and low symmetry will it be necessary to use a bin sort and direct access disk, and so for most calculations the reordering will not be I/O bound on the CRAY-2.

For high symmetries (which normally give multicentered symmetry orbitals) and small molecules the only sparseness in the integral list will come from symmetry. However, for lower symmetries and larger systems (that is, for larger distances between basis functions) there may be increasing sparseness in the integral list from the neglect of very small integrals in MOLECULE. It will often be useful to exploit such sparseness to reduce the length of the ordered integral list, as well as the raw integral files. On the other hand, if this is done simply by adding a label word to each integral the sparseness must be greater than a 50% reduction in order to see any reduction in the length of the list. Instead, the index locating a nonzero integral in the full list can be packed into the lowest bits of the integral itself. As each buffer of ordered integrals fixes one pair index *ij*, it is only necessary to pack the *kl* pair index, for which it is convenient to use 16 bits. This involves a loss of between four and five decimal places in the mantissa, which is acceptable for most applications, given the 14-digit precision on Cray computers. The examination of the ordered integrals, prior to writing them out, to check whether their magnitude exceeds some threshold, can be viewed as building a GATHER pointer vector (strictly, a COMPRESS pointer): the subsequent GATHER and the packing of the GATHER pointer elements into the lowest bits of the integrals can then be completely vectorized. The entire process can be viewed as a "compressed index" vector operation on the X-MP,[11] and under CFT this can be vectorized by the compiler without any need to allocate space for the GATHER index vector. The subsequent processing of such "compressed" integrals is discussed below.

Timing for ordering various integral files is given in Table 8 for N_2 calculations on the X-MP/48. When spherical harmonics are used in D_{2h} symmetry, the ordering can be carried out entirely within 2 MW of

memory. This is another useful trade-off against integral generation time when using spherical harmonics: the ordering takes five times longer overall when Cartesians are used, and unless about 3.5 MW of memory is available requires either direct access disk or SSD space. It should be noted that a request for this memory would incur a priority penalty on our X-MP/48, according to the "$1/n$" rule (see Section 3.4 and Ref. 28). When ordering in memory up to 1,000,000 ordered integrals can be produced per second, while for out-of-memory cases the observed best rate depends on the symmetry. In high symmetry cases about 300,000 ordered integrals can be produced in one second, but for the low symmetries this rate rises to more than 500,000 per second. Table 9 contains results for integral ordering obtained on various Cray computers. All of the results given are for sorting in memory. The Y-MP results are somewhat faster than would be expected from the ratio of Y-MP and X-MP clock periods, although the speed increase is not as great as was observed for the integral calculation. The CRAY-2 is considerably slower in this step than the X-MP, because of its much slower memory.

As we shall see, for some applications it is desirable to have a different combination of integrals, such as the "$\mathscr{P}$-supermatrix" with elements

$$\mathscr{P}(ij\,|\,kl) = (ij\,|\,kl) - \tfrac{1}{4}[(ik\,|\,jl) + (il\,|\,jk)] \tag{7}$$

For later convenience the "diagonal" elements $ij = kl$ are usually halved. The reprocessing of the ordered integrals to obtain these values can be handled using very similar techniques to those described above. In fact, as fewer symmetry blocks of $\mathscr{P}$ are usually required than of the integrals themselves, the processing is even simpler.

4.3. SCF Calculations

The most time-consuming part of most SCF calculations is the contraction of integrals with density matrix elements to build one or more Fock matrices. For the closed-shell Fock operator F this process can be written as

$$F_{ij} = \sum_k \sum_l \mathscr{P}(ij\,|\,kl) D_{kl} \tag{8}$$

where D is the density matrix and $\mathscr{P}$ is the supermatrix introduced in the previous section.[53] Operationally, however, this step is driven by reading blocks of $\mathscr{P}$ from file: blocks corresponding to kl values for fixed ij such that $kl \leqslant ij$ give rise to two contributions:

```
      DO 10 KL=1,NKL
         F(KL)=F(KL)+D(IJ)*P(KL)
10    CONTINUE
      DO 20 KL=1,NKL
         F(IJ)=F(IJ)+D(KL)*P(KL)
20    CONTINUE
```

The first of these loops is clearly a SAXPY operation and the second is a dot product. Hence these steps are immediately vectorizable. Further, if the vector **P** has been compressed to only nonzero values these two steps can be viewed as a sparse SAXPY and a sparse dot product, and are again vectorizable as described in Section 3.3 above. A more complete discussion of this aspect can be found in Refs. 28 and 54.

For open-shell energy expressions,[53, 55, 56] additional Fock operators must be built from other combinations of integrals, such as the $\mathscr{K}$ supermatrix,

$$\mathscr{K}(ij|kl)=\tfrac{1}{2}[(ik|jl)+(il|jk)] \tag{9}$$

and open-shell density matrices. Such work is obviously vectorizable in complete analogy with the $\mathscr{P}$ processing described above. If multiple open-shell densities are employed, the vectors of K values can be reused from vector registers, enhancing performance. Alternatively, it is possible to read the $\mathscr{P}$ and $\mathscr{K}$ supermatrices in separate tasks, thereby utilizing more than one CPU simultaneously. It is also possible to use multiple CPUs to process different parts of the same supermatrix, as discussed in Ref. 28. However, if no sparseness is employed, the loops for constructing F run at 60–100 MFLOPS on the X-MP, and therefore (running the two loops sequentially) require reading up to 25 MW/s from file to avoid becoming I/O bound. Any tactic to increase the effective rate of CPU operations, such as using multiple CPUs, obviously only increases the demands on the I/O system. One possible solution, as discussed previously, is to keep the supermatrix files on the SSD; another, discussed in more detail below, is to keep the files in memory on the CRAY-2.

The only other step in an SCF calculation that consumes anything but trivial amounts of time is the Fock matrix diagonalization, and for most applications with up to 200 basis functions and symmetry blocking this step is quite inexpensive. Although specially optimized matrix diagonalization routines (such as the EISPACK library[57]) are available in CRI's SCILIB library, the requirements for SCF calculations are usually so modest that simple Jacobi or Givens schemes are adequate.

The fraction of the SCF time required for Fock matrix construction is illustrated by the N_2 results of Tables 8 and 9. For the high symmetry

calculations 50% or more of the SCF iteration time is used in this step. For the low symmetry calculations the diagonalization of the Fock matrix is relatively more time-consuming. For the D_{2h} case the Y-MP results of Table 9 are essentially what would be expected from its clock period, while the CRAY-2 results are rather poor, given that the sparse dot product and SAXPY performance is expected to be similar to that of the X-MP (see Section 3.3). However, the X-MP and Y-MP versions of the SCF code read $\mathscr{P}$ in large fixed-length blocks, so part of the difference between these machines and the CRAY-2 may result from I/O overheads on the latter.

4.4. Integral Transformation

The treatment of electron correlation requires that the integrals be transformed from the atomic orbital basis set to the molecular orbital basis set. The computational aspects of this step have been extensively reviewed by Wilson.[58] Transformation of integrals can be implicit in some approaches, such as the method of self-consistent electron pairs,[59, 60] or explicit, involving reading the file containing the AO integrals, transforming these to the MO basis, and writing a new file containing the MO integrals. Hybrid methods for treating the correlation problem are also possible, where some contributions are computed in the MO basis set and some directly from the AO integrals, thus requiring only a partial transformation.

Without symmetry, the full transformation is of the order mn^4, where n is the number of basis functions in the AO basis set and m is the number of MOs. First, it is obviously beneficial to minimize m, and the simplest step in this direction is to eliminate transforming to any orbitals that are not occupied in the correlation treatment. Furthermore, the contribution to the energy arising from electrons that are not correlated (the core) is straightforwardly expressed in terms of Coulomb and exchange integrals only (see, e.g., Ref. 61). This is commonly accounted for by constructing a core Fock operator to be added to the one-electron integrals. In our implementation, this core Fock operator can be constructed using the supermatrix formulation described above for the SCF procedure, or directly from the ordered integrals without further sorting. Clearly, constructing the Fock operator from the supermatrix is faster provided the supermatrix already exists. However, if the supermatrix is not available, the work required to construct one Fock operator directly from the ordered integrals is roughly equal to the time to form the supermatrix—this cost can be amortized over the iterations of an SCF calculation as the supermatrix is used a number of times, but for the core Fock operator construction we prefer to avoid the extra I/O of the sorting step and to construct the operator from the ordered integrals.

In addition to the savings from avoiding transforming for core or deleted virtual orbitals, the overall work can be reduced by exploiting sparseness in the integral or coefficient arrays. While some reduction can be effected simply using sparse vector or matrix arithmetic, in cases where the sparseness derives from the symmetry of the system it is preferable to handle the symmetry explicitly. The use of symmetry in the transformation can reduce the overall work by orders of magnitude: if a basis set of n functions is symmetry adapted with $n/2$ basis functions in each of two irreducible representations, the work is reduced from one transformation of order n^5 to four transformations of $n^5/32$, or an eightfold reduction. Clearly, for systems with higher symmetry, such as D_{2h}, the savings would be even larger. It should be noted that the operation counts given here are based on the assumption that the two-electron integrals are available in a basis of symmetry orbitals. While there are schemes for obtaining the advantages of high symmetry in the SCF step without the formation of symmetry integrals,[62–64] such schemes become very complicated for the transformation. However, as described above, the symmetry transformation in the calculation of the integrals is very efficient, and the very large savings in the transformation (and SCF) steps that derive from it are real, not a consequence of moving work from the transformation into the integral evaluation. Of course, the simple and efficient symmetry-adaptation procedure described in Section 4.2 is restricted to D_{2h} and its subgroups (although see Ref. 50), but very few treatments of electron correlation are implemented in higher symmetries anyway.

We now discuss our implementation of the transformation, starting with the one-electron integrals. While this requires very little of the overall time, it illustrates several aspects of the transformation of the two-electron integrals. The transformation of the one-electron integrals, H, by the coefficient matrix, C, can be written as

$$\mathsf{H}' = \mathsf{C}^T\mathsf{H}\mathsf{C} \tag{10}$$

but the one-electron integrals H_{ij} are symmetric and therefore naturally stored as $i \geqslant j$. If the integrals are stored only as this lower triangle, it is difficult to vectorize (10). However, if the integrals are expanded to a square matrix (a matrix T) the transformation (10) clearly becomes two matrix multiplications. The transformed integrals naturally form a square matrix, which can be compressed to a lower triangle before being written to disk. Based on the timings reported in Section 3 above it is clear that a formulation in terms of matrix multiplications will be very efficient on Cray computers. Hence if the squaring of the AO integral matrix or the compression of the transformed integrals is coded inefficiently, these

processes could actually become the rate-limiting step. That is, the pre- and post-processing of the integrals, which are steps of order n^2, could take longer than the n^3 matrix multiplication step, at least for any n that is likely to be encountered in present quantum chemical calculations. Two efficient approaches of squaring (or of compressing) the matrix can be considered. In the first, a row is moved from lower triangular storage to a row and column in the square form:

```
      IX=0
      DO 10 I=1,N
         DO 20 J=1,I
            T(J,I)=H(J+IX)
            T(I,J)=H(J+IX)
20       CONTINUE
         IX=IX+I
10    CONTINUE
```

In the second approach a vector of length n^2 is used to hold a mapping into each location in T from a location in H. Then by using a GATHER the matrix T can be easily formed. Similar procedures can be used for the compress. (Note that many GATHER operations can be viewed alternately as SCATTER operations: we use the term GATHER to describe either coding organization). In this approach there is obviously additional overhead in forming the GATHER pointers, but in the transformation of the two-electron integrals the same steps occur so often that the cost of constructing the GATHER pointers once at the beginning of the calculation is effectively amortized to zero.

Either approach to squaring a one-electron integral matrix works well on Cray computers. However, on a machine with a large overhead associated with starting vector operations, like the CDC CYBER 205 or some of the Japanese supercomputers, the GATHER implementation is to be preferred. We have therefore implemented the GATHER approach since it gives us some additional portability.

For large basis sets the two-electron integral transformation poses some problems in data handling. Yoshimine[65] has described an efficient process for transforming two-electron integrals $(pq|rs)$ into $(ij|kl)$ that requires only a small fraction of either $(pq|rs)$or $(ij|kl)$ to be in core at one time, that is, an out-of-core transformation. In successive steps a list of $(pq|rs)$ is converted into a list of $(ij|rs)$, which is then transposed to a list of $(rs|ij)$ and transformed to a list of $(kl|ij)$. If the ordered integrals are stored such that for each $r \geqslant s$ all $p \geqslant q$ values are present (this requires some double storage in the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ symmetry blocks) it is possible

to read in the "*rs* row" of the integrals, square the *pq* indices and perform a two-index transformation on this subset of the integrals analogous to the one-electron integrals. This set of transformed integrals $(rs|ij)$ comprises all *ij* for fixed *rs*, and can be compressed to distinct integrals $(i \geqslant j)$ only before being written to disk. We also compress the final transformed integrals before writing them to file such that only the unique values are stored. That is, any double storage in the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ blocks is removed.

Some additional considerations that were not encountered in the one-electron transformation affect the coding of the two-electron case. For example, the number of two-electron integrals can be very large and for extended systems and low symmetries there may be many accidental zeroes. As noted in Section 4.2 we have the option to store only those symmetry orbital integrals above a given threshold: each *rs* row of integrals is compressed to an array of nonzero integrals with the position index of each integral in the uncompressed row packed into its low-order 16 bits, these are written out with the number of nonzero *pq* values. To expand these values to the full row we clear an array the length of the full row, mask off the low-order bits into an integer array, and use this integer array as the pointer vector in a SCATTER operation. All of these steps are vectorizable, so the process is very efficient. Another aspect of the transformation concerns the appearance of the transposed coefficient matrix $\mathbf{C}^T$ in (10) above. Based on the observations presented in Section 3.2, it is better to transpose a matrix explicitly and use the SCILIB routine MXM than to use MXMA to multiply by the transpose, as this minimizes problems with bank conflicts on the CRAY-2. Of course, in the two-electron transformation the same $\mathbf{C}$ and $\mathbf{C}^T$ matrices are used repeatedly, so there is no loss in efficiency (and only a trivial increase in storage) if we store both $\mathbf{C}$ and $\mathbf{C}^T$ and use MXM. Where the integral list is sparse because of accidental zeroes it would be advantageous to use a sparse matrix multiplication routine like Saunders' MXMB,[18] but at present we use only the Cray library routines.

While formally the two-electron transformation process can be considered as two series of two-index transformations, another complication relative to the one-electron case is that a transposition of the results is required before the second series of two-index transformations. In our implementation the transposition is merged into the two-index transformations. That is, as each two-index transformation is completed the resulting integrals are moved into bins for the direct access I/O. It is important to note that this process is actually a transposition and not a sort, so there is no index calculation: the first *w* elements go into bin 1, the second *w* into bin 2, etc; the value of *w* is determined by the available memory. This step can be coded as a series of vector moves into the bins. The transposition step is also merged with the second half-transformation: after the contents of a chain of bins is SCATTERed into memory, all related two-index trans-

formations are performed and the fully transformed integrals are written to disk. The process is then repeated for the next chain of half-transformed integral bins.

Another special circumstance in the two-electron transformation arises for symmetry types $\alpha\beta\alpha\beta$ and $\alpha\beta\gamma\delta$, since there is no permutational symmetry between r and s or between p and q in integrals $(pq|rs)$ of these symmetry types. There is therefore no need to square raw integrals or to compress transformed integrals in these cases. However, handling the different cases does not give rise to any noticeable overhead, as we separate the various symmetry types at a very high level (see Section 4.2 above). Overall, the inclusion of symmetry involves rather straightforward coding and introduces little overhead, but provides enormous gains in performance.

Two-electron integral transformation timing data for our N_2 example are given in Table 9. The CPU times are effectively negligible in comparison with other steps (notably the integral evaluation) on all machines. As ordered integrals are used as input, and only a transposition is required after the first half-transformed step, the wall clock times are also very small. The CPU times given correspond to a rate of around 50 MFLOPS on the X-MP and CRAY-2, and 65 MFLOPS on the Y-MP. There is actually rather substantial overhead in these times: if a transformation is performed for the C_1 symmetry N_2 calculation described in Section 4.2 the X-MP/48 CPU time required is 324 s, but the rate increases to 150 MFLOPS. Thus the use of D_{2h} symmetry in this case gains a factor of 38 in CPU time, but there is a concomitant threefold loss in performance because of the overheads involved in processing numerous rather small symmetry blocks. For a very large basis, in which the transformation overheads would be amortized to nearly zero, the use of D_{2h} symmetry would give an improvement of more than two orders of magnitude over the no symmetry case.

As expected, the out-of-core transformation described here works very well on Cray computers. However, on some other computers, such as the CDC CYBER 205, the large vector start-up overhead leads to poor performance for even the larger matrices. Therefore we have also programmed an in-core transformation. We use a modification of the Bender approach.[66] As in the out-of-core scheme we separate those cases with and without rs permutational symmetry. The first half-transformation is replaced by two matrix multiplications. By ignoring the permutational symmetry between pq and rs (that is, the $\alpha\alpha\alpha\alpha$ case is treated as $\alpha\alpha\beta\beta$ and the $\alpha\beta\alpha\beta$ case as $\alpha\beta\gamma\delta$), the second half-transformation can be written as long SAXPY operations. The redundant integrals for the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ cases are eliminated using a precomputed GATHER pointer vector. This organization yields excellent performance on the CYBER 205. As this approach eliminates the transposition and I/O, it also yields essentially the same

performance as the out-of-core approach on all Cray computers, for those calculations for which the transformed integrals can be held in core.

By having both an in- and out-of-core transformation we thus have some additional portability without sacrificing any performance. The program described is so highly vectorized that the n^5 transformation step commonly represents an insignificant fraction of the total time. This is very different from the situation on scalar machines.

4.5. Full Configuration Interaction

The full CI (FCI) procedure, the use of all configurations that can be constructed from a given one-particle basis set, is an exact solution to the correlation problem for this basis set, and therefore stands as an unambiguous test of approximate methods. Because of recent advances in full CI methodology (principally in vectorization) and developments in computer hardware it has been possible to perform an important series of benchmark calculations.[67, 68] Of course, it is still not possible to perform FCI calculations for large basis sets and problems of chemical interest, but it is nevertheless instructive to begin our discussion of correlated wave function generation with the FCI method. Its simplicity makes it ideal for explaining some of the concepts of vectorizing other approaches, and FCI wave functions in a limited one-particle space lie at the heart of the CASSCF method. We therefore consider the FCI approach first and then proceed to the other methods.

In most modern CI calculations, with any type of configuration space, the Hamiltonian matrix is too large to be held in memory, and therefore an iterative diagonalization procedure is required to obtain the CI energy eigenvalue(s) and eigenvector(s). This is true even on the CRAY-2 when configuration spaces of order 10^6 and more are to be handled. The Davidson diagonalization procedure[69, 70] is most commonly used. This method has several advantages, one of which is that it imposes no constraints on the order in which matrix elements are processed. This allows the matrix elements to be computed in an order that achieves the maximum overall performance, whereas techniques that implicitly require access to, say, one row of the matrix at a time might lead to intolerable degradation of performance. The key is thus to form efficiently the residual vector σ, the product of the current estimate of the CI eigenvector $\mathbf{c}$ and the Hamiltonian matrix H. This step can be written as

$$\sigma_K = \sum_p \sum_q \sum_r \sum_s \sum_L (pq|rs) B^{KL}_{pqrs} c_L \tag{11}$$

where B are the "coupling coefficients." Siegbahn pointed out[71] that if

the known factorization of B^{KL}_{pqrs} into products of one-electron coupling coefficients (using a resolution of the identity),

$$B^{KL}_{pqrs} = \sum_J A^{KJ}_{pq} A^{JL}_{rs} \tag{12}$$

is explicitly inserted into (11), it is possible to vectorize the calculation of σ very straightforwardly. First, the product of one set of coupling coefficients and **c** is collected in D,

$$D^J_{rs} = \sum_L A^{JL}_{rs} c_L \tag{13}$$

This can be implemented as a set of SAXPY operations involving the elements of the very sparse matrix A. A matrix multiplication of the intermediate array D and a block of the integrals is performed,

$$E^J_{pq} = \sum_{rs} (pq|rs)\, D^J_{rs} \tag{14}$$

The intermediate array E is then contracted with the second set of coupling coefficients to form a contribution to σ,

$$\sigma_K = \sum_J \sum_{pq} A^{KJ}_{pq} E^J_{pq} \tag{15}$$

This operation is a matrix–vector product. It is clear that all steps in the calculation of σ can be vectorized, given that all the necessary quantities are available as needed. This is easily arranged for σ, **c**, and the integrals (the latter can be stored in memory for any MO space it is feasible to use in a full CI calculation). The handling of the coupling coefficients A requires more attention. As there is a rather large number of A^{KL}_{pq} elements these must either be precomputed and stored on disk [sorted to an order which allows vectorization of (13) and (15)] or computed on the fly as required in these two equations.

Siegbahn originally introduced this method[71] to improve the performance and increase the size of the FCI step in CASSCF calculations. In this context it was considered appropriate to use spin eigenfunctions as the configuration basis and to compute and store a list of the coupling coefficients on disk. This would require a great deal of disk space for large calculations, but is acceptable for calculations with up to, say, 12 electrons and 12 orbitals in the FCI. For larger calculations, disk space would become excessive, even though the CPU time requirements would remain acceptable, so that disk space becomes the limiting factor. Knowles and Handy[72] showed that by using determinants instead of spin eigenfunctions

as the configuration basis the nonzero values of A_{pq}^{KL} (all of which then become ± 1) can be computed on the fly. The CI vector **c** is several times longer in a determinantal basis, but this is no handicap on computers such as the CRAY-2. In this way FCI calculations can be performed using very large expansions, providing a means of benchmarking approximate methods as well as performing large CASSCF calculations. The trade-off of memory (and, perhaps, CPU time) for disk storage is, of course, not an uncommon feature of programming modern supercomputers: we discuss below other trade-offs that would not have been considered a few years ago when CPU power was the limiting factor.

As the most time-consuming step in the FCI algorithm (13)–(15) is a matrix multiplication, the code is very efficient on Cray computers. This has allowed calculations as large as 28,000,000 determinants to be performed on the CRAY-2.[73] Such a calculation requires on the order of 100 min of CRAY-2 CPU time per FCI iteration: something over 90% of this time is spent in matrix multiplication. As **c**, σ, and some scratch arrays must be held in core, the memory required for such a calculation is about 60 MW, which is perfectly feasible on the CRAY-2. However, the Davidson diagonalization process requires the **c** and σ vectors from the previous iterations. Therefore, just as storing the coupling coefficients can exhaust the disk storage long before the CPU time become prohibitive, so can the need to store the previous **c** and σ vectors. As Davidson noted,[69] it is possible to "fold" all the previous vectors into one vector, effectively starting again with the current estimate of the eigenvector as the starting guess. While this can slow convergence somewhat, it can reduce considerably the disk storage required. In effect, the CPU performance and the high degree of vectorization, compared to the disk performance and space limits, mandate implementing the folding procedure if the largest possible FCI expansions are to be considered.

In addition to the larger dimension of the FCI problem using a determinantal basis, there is another difficulty: the potential collapse of a desired higher root of the secular problem to a lower root of a different spin symmetry due to numerical rounding.[74] This can be a severe problem in the application to CASSCF wave functions, where many roots of the secular problem may be required in investigating problems related to spectroscopy. Recently, Malmqvist *et al.*[75] have developed a method for using spin eigenfunctions of the desired symmetry (configuration state functions, CSFs) instead of determinants, but without the need to store the coupling coefficients on disk. In their approach the graphical unitary group representation of the CSFs is used (see, e.g., Ref. 76). The graph that defines the configuration space is split into an upper and lower portion, and at each graph node on the dividing line contributions to the product (13) are computed and accumulated using matrix multiplication. This

requires subsets of coupling coefficients A_{pq}^{KL} for each portion of the graph, plus some "partial coefficients" for cross-terms between the two portions. All of this coupling coefficient information can be held in memory, even for large configuration spaces (100,000 CSFs and more). Such an approach seems ideal for the CASSCF problem, where the number of active orbitals is limited, but may not be suitable for the large calculations used for benchmarking. It is clear that this is currently an active area of research, and as yet the best method of performing FCI calculations may not have been achieved. However, even the current approaches illustrate the very high level of vectorization that can be achieved in CI methods for solving the correlation problem. They also illustrate the need for careful thought about trade-offs in disk storage, memory use, and CPU time in order to maximize the size of problem that can be solved.

4.6. Symbolic Formula Tape for Multireference CI Calculations

The most general type of large-scale CI wave functions in current use is that comprising a set of reference CSFs and all CSFs singly and doubly excited with respect to these reference CSFs. The first task in generating such a wave function is to evaluate the coupling coefficients involved in the various Hamiltonian matrix elements. We begin by classifying the MOs into inactive, active, and secondary orbitals. The inactive orbitals are doubly occupied in all reference CSFs (but can have other occupations in the CI configuration space—orbitals that are always doubly occupied are absorbed into the effective one-particle Hamiltonian as described in Section 4.4). The active orbitals have different occupations in different reference CSFs and the secondary orbitals are unoccupied in the reference CSFs. Classes of configurations can be defined by their *internal occupation*, or internal for short. An internal occupation consists of a particular sequence of active and inactive orbitals containing n, $n-1$, or $n-2$ electrons altogether. Those internal occupations that contain n electrons give rise to valence configurations simply by enumerating the possible spin-couplings with which they can appear; valence configurations must have the same spatial symmetry as the desired CI root and must differ from a reference occupation by no more than a double excitation. Internals with $n-1$ or $n-2$ electrons are incomplete, in the sense that to obtain CSFs it is necessary to add one or two secondary orbitals to the occupation. Evidently, a given $n-2$ internal, say, can generate a set of CSFs by coupling in different pairs of secondary orbitals with different spin-couplings. Again, the resulting n-electron occupations are constrained to differ from the reference occupations by no more than a double excitation. By expressing a CI configuration list in this manner it becomes clear that the same coupling coefficient applies to a large set of matrix elements; the

coupling coefficients are determined essentially by the internals from which the CSFs are derived, so that the particular secondary orbitals that appear are irrelevant. Therefore, if the coupling coefficients are computed for all possible internal–internal interactions, all of the required CI matrix elements can be computed using the appropriate integrals and these coupling coefficients.

In the first direct CI programs the unique coupling coefficients were computed by hand and coded into the program.[77] The types of wave function for which this is feasible are very restricted, and programs following this approach were available only for single reference configuration wave functions based on a closed-shell determinant or a UHF determinant. The direct CI approach became much more general once Siegbahn[78, 79] combined the factorization of CSFs (and of coupling coefficients) into internal and external parts with the graphical unitary group approach as formulated by Shavitt.[80] Several variations on this idea were implemented, but most suffered from some limitations in the calculation of the coupling coefficients, a step formulated as essentially a set of scalar operations. Once the coupling coefficients were evaluated, on the other hand, the calculation of the eigenvalues was very efficient, just as in the FCI case, and we discuss this aspect in more detail in the next subsection. The problem in the evaluation of the coupling coefficients was most severe in the case that the number of references actually used was a small subset of those possible for a given choice of active and inactive spaces, which is unfortunately a rather common case. One alternative approach would be to avoid the unitary group and use a more conventional CI approach. The internal occupations are generated, together with all possible spin-couplings to which they give rise (including coupling of one or two electrons in model external orbitals if required). The occupations are compared to identify potentially nonzero matrix elements, and the coupling coefficients evaluated between prototype CSFs represented by the individual spin-couplings. Such an approach has been implemented,[81] but although it can avoid some of the problems associated with the unitary group approach, the calculation of the coupling coefficients is still not vectorizable.

Recent work along quite different lines by Knowles and Werner[82] and by Siegbahn[83] has enormously simplified the calculation of the coupling coefficients. This new approach again exploits the factorization of the two-electron coupling coefficients into sums of products of one-electron coupling coefficients, as in (12). First, the internal occupations are generated. In computing the one-electron coupling coefficients a product form is used: a fictitious MO a is introduced, which is unoccupied in all the internals, whereupon we can write

$$A_{pq}^{KL} = \sum_{M} A_{pa}^{KM} A_{aq}^{ML} \tag{16}$$

Table 10. CI Timings for N_2 on CRAY X-MP/48 (in seconds)[a]

Calculation	Time
Formula tape for 16 internals[b]	<0.1
Direct CI iteration (17 626 CSFs)	0.9
Formula tape for 2804 internals[c]	12.5
Direct CI iteration (729 950 CSFs)	79.7
Processing $(ai\|bj)$ integrals	38.3
Processing $(ai\|jk)$ integrals	18.1
Processing $(ab\|cd)$ integrals	10.2

[a] [5*s*4*p*3*d*2*f*1*g*] basis: $1\sigma_g$ and $1\sigma_u$ MOs frozen.
[b] Single reference CSF, 10 electrons correlated.
[c] CAS reference space (6 active electrons in 6 orbitals—32 CSFs), 10 electrons correlated.

where M is an "internal" to which a has been coupled. The A_{pa}^{KM} values are very simple to calculate and the values can be held in memory. In evaluating the two-electron coupling coefficients, pairs of internals are compared and the type of coupling coefficients that arise are identified (these types are determined by the orbital differences between the pair and the open shells they contain). The coupling coefficients between all CSFs that can be derived from the pair of internals are then evaluated by matrix multiplication of the stored one-electron terms. The corresponding formula tape entry comprises the internal labels, the label of the integrals (or blocks of integrals) that appear, and the coupling coefficients. In this approach there is little redundant work even where only a few of the possible reference occupations arising from a given active space are used. Furthermore, unlike the approach using prototype CSFs described above, the evaluation of the coupling coefficients is highly vectorized. The program that we use was developed by Siegbahn[83] and performance better than 50 MFLOPS has been observed on the X-MP/48. Some timing results are given in Table 10. For our N_2 example the coupling coefficients for single and double excitations from a single closed-shell reference CSF (16 internal occupations) require much less than one CPU second, while for the case of a CAS reference space that gives rise to 2804 internals the coupling coefficients require 12.5 s. These very recent developments means that the rate-limiting step in an MRCI calculation is in the calculation of the eigenvector, and the excellent performance of the new approach allows very large reference spaces: the calculation of the coupling coefficients for more than 60,000 internals requires about 700 s on the X-MP/48.

4.7. Multireference CI Eigenvalue Determination

The FCI calculations described above show that it is possible to achieve very high performance in the eigenvalue determination through the formulation of the time-consuming step in terms of matrix multiplication. The factorization into internal and external contributions in the multireference direct CI (MRCI) case, with the consequent association of a set of CSFs and an array of CI coefficients with each internal occupation, is also very well suited to a matrix multiplication formulation,[81, 84–86] and so again very high performance should be possible. On the other hand, there are substantial differences between the FCI and MRCI cases, resulting mainly from the fact that all orbitals in the FCI calculation are classified as part of the same orbital space and their number is so small that all integrals can be held in memory, while in the MRCI calculations there is a distinction between occupied and secondary orbitals, and it is seldom possible to hold all the integrals in memory. For MRCI calculations, additional data-handling involving sorting both integrals and coupling coefficients must thus be performed. If $i, j, k\ldots$ are active or inactive orbitals and $a, b, c\ldots$ are secondary orbitals, the types of interactions between internal occupations can be classified by the type of integrals required: $(i|h|j)$, $(a|h|i)$, $(a|h|b)$, $(ij|kl)$, $(ai|jk)$, $(ab|ij)$, $(ai|bj)$, $(ab|ci)$, and $(ab|cd)$. Each class of integrals contributes to a limited number of types of interaction. Therefore the determination of the eigenvalue first involves sorting the integrals into classes, and to a given order within classes. The formula file is sorted during its generation to have the same organization of types as the integral file. In each MRCI iteration the program loops over each type of interaction and within each type a block of coupling coefficients is read. If these coupling coefficients refer to a new block of integrals, these are also read in. In our organization, the formulas and integrals are ordered so that the integral and formula files are read only once per iteration, and of the order of N_{MO}^2 integrals are required to be in memory at one time, where N_{MO} is the number of MOs.

Each possible interaction is treated in a subroutine specific to the integral class: the details of the treatment have been extensively reviewed by Saunders and van Lenthe,[86] and we will not repeat them here. As shown in Ref. 86, most types of interaction can be reduced to multiplying a block of integrals by coupling coefficients, and then a subsequent matrix multiplication of the results by a block of CI coefficients. (In practice, the block of integrals may actually be a sum of two different blocks multiplied by two coupling coefficients.) The final matrix product is added to a block of the σ vector, so there is some saving of memory if the matrix multiplication routine allows the product to be added to (or subtracted from) the destination array.

Symmetry is used in all sections of the CI code, just as in the transformation. This includes the index permutation symmetry of the integrals as well as the spatial symmetry. If P indexes a particular $(n-2)$-electron internal, for example, the coefficients of the distinct double excitations generated by coupling to P can be written as an array

$$c_P^{ab} \forall\, a \geqslant b \tag{17}$$

Such arrays display the same permutational symmetry as one-electron integrals: if a and b are of the same symmetry type the array is triangular, while if they are of different symmetries the array will be square. Some contributions to σ will then be required only as lower triangles, as the blocks of σ have the same structure as those of **c**. In such cases we must square the block of integrals and CI coefficients, in the same manner as in the transformation, form the necessary product, and then fold the two halves of the product matrix together to add to the σ vector. These squaring and folding operations are vectorizable, as discussed in Section 4.4 above, and take only a small amount of time. Therefore, while an MRCI calculation involves a far more complex organization than the FCI scheme discussed in Section 4.5, each individual step is vectorizable and very high performance can be obtained. We have been able to routinely perform calculations involving 1 million CSFs and have on occasion performed calculations involving more than 4 million CSFs.

Some timing results are presented in Table 10. For N_2 with the 10 valence electrons correlated and an SCF reference the direct CI iteration time is less than 1 s on the X-MP/48. This calculation involves 17626 CSFs. The observed performance is about 44 MFLOPS: lowering the symmetry to C_{2v} doubles the iteration time but also improves the performance to 66 MFLOPS. In the limit of no symmetry about 140 MFLOPS would be obtained. Hence for very large basis sets, in which the overheads have become negligible, we may expect single reference direct CI performance of about 140 MFLOPS on the X-MP/48. In multireference cases, the performance is somewhat higher. For example, timing for an N_2 calculation with a CAS reference space (six active electrons in six active orbitals, giving 32 reference CSFs) is also given in Table 10. The iteration time of about 80 s for almost 730,000 CSFs corresponds to an average of 50 MFLOPS. Hence in very large (or low symmetry) cases we may expect MRCI performance of better than 150 MFLOPS. A breakdown of that part of the iteration time concerned with the three most time-consuming classes of integrals processed is also given in Table 10. The $(ai|bj)$ integrals [this actually includes $(ab|ij)$ integrals as well] require the most effort, followed in this case by $(ai|jk)$ and $(ab|cd)$. For very large basis sets the latter are expected to dominate, but this case is rarely observed in practice. Finally, we note

that on the CRAY-2 the observed times are up to twice as long as on the X-MP/48: these N_2 calculations involve multiplication of rather small matrices and the matrix multiplication performance on the CRAY-2 will suffer somewhat. On the other hand, the iteration times on the Y-MP are about 2/3 of those on the X-MP, suggesting that large MRCI calculations will run at well over 200 MFLOPS on the Y-MP.

4.8. CASSCF Calculations

The CASSCF approach can account for near-degeneracy correlation effects and correlation contributions that vary rapidly with geometry.[87] It can thus supply an excellent zeroth-order description for use in the MRCI approach. We employ a two-step, uncoupled MCSCF optimization scheme,[88] which involves a transformation of the integrals, determination of an FCI wave function, construction of a gradient vector and Hessian matrix for the energy dependence on orbital rotations, and solution of the simultaneous linear equations of the Newton–Raphson method for obtaining improved orbitals. In practice, far from convergence it is better to use a first-order approach for optimizing the orbitals,[89, 90] in which case an approximate Hamiltonian matrix over single excitations must be constructed instead of the Hessian; this Hamiltonian matrix is then diagonalized in order to obtain improved orbitals. We now consider each of these steps in more detail.

Just as for the MRCI case discussed in the previous subsection, different classes of integrals are required in a CASSCF calculation. For the CI step, only integrals with four active orbital indices are required (the inactive orbital contributions can be absorbed into the one-electron operator, as discussed in Section 4.4 above), while for the gradient of the energy with respect to orbital rotations, integrals with three active orbital indices and one index that runs over the full MO space are required. Finally, for the orbital rotation Hessian, integrals with two indices running over the full space and two over the space of occupied (inactive and active) orbitals are required. We compute all integrals required for a given MCSCF iteration in the same transformation step. The two-electron integral transformation is performed in a manner very similar to the full two-electron transformation described above in Section 4.4, but there are some special features added because only certain classes of transformed integrals are required. For example, if $m, n \ldots$ denote occupied MOs and $p, q \ldots$ all MOs, it is clearly advantageous to obtain integrals $(pq|mn)$ and $(pm|qn)$ by first transforming integrals over index n and then over q. In this way no step in the transformation scales worse than MN^4, for N AOs and M occupied MOs, an N/M-fold reduction over the full transformation. Further, the ranges used in the second half-transformation can obviously

be modified depending on which MO indices appear in the half-transformed integrals. Some care is needed in this approach when symmetry is used. Thus if the integrals $(p_\alpha q_\alpha | m_\beta n_\beta)$ are required, where α and β label irreducible representations, it would be necessary to perform a full transformation on the α symmetry indices at the outset if the $\alpha\alpha\beta\beta$ AO integral block is to be processed only once. Instead, following Roos,[91] this block is processed twice in the first half-transformation, once as $\alpha\alpha\beta\beta$ and once as $\beta\beta\alpha\alpha$. In this way the first transformation step always scales as MN^4. The same strategy is employed for the $\alpha\beta\gamma\delta$ blocks as well.

The FCI calculation is handled using the factorized coupling coefficient approach of Siegbahn,[71] as described in Section 4.5. The construction of the orbital Hessian requires a very small amount of time and in our current version is still scalar code. The solution of the simultaneous equations is performed using an iterative scheme analogous to the Davidson diagonalization.[69] As we are normally able to store the full orbital Hessian in memory, the matrix–vector product needed in the iterations can be performed one row at a time; it is then identical to the construction of the Fock matrix [equation (8) above] and is implemented as a SAXPY and a dot product. (The handling of the approximate Hamiltonian matrix appearing in the first-order optimization scheme is very similar.) Note that for very large cases (or in the absence of symmetry) this step could also proceed as for the Fock matrix construction by storing the ordered rows on disk and rereading them in each iteration. If necessary, the same compression techniques as used for the SCF supermatrices could be employed, together with a sparse SAXPY and dot product.

The time-consuming steps in a CASSCF calculation are thus vectorizable, and using these techniques we have been able to perform calculations involving about 40,000 CSFs and a basis set of about 200 functions. For the N_2 example we have used in these discussions the CASSCF iteration time is dominated by the integral transformation (which takes only a few seconds on the X-MP/48), the orbital optimization step requires one second, and the CI time is negligible (and is anyway almost entirely overhead for this case—32 CSFs). For a configuration space of some 3500 CSFs the CI step requires about 5 s per direct CI iteration, while spaces of 40,000 CSFs would require about 80 s on the X-MP.

4.9. Avoiding I/O, Direct Methods

One of the constraints on how efficiently calculations can be performed, as we have repeatedly noted, is the rate at which data can be retrieved from secondary storage. SSDs provide a partial answer to this problem for X-MP (and Y-MP) machines, but these devices are not available for the CRAY-2. However, the large central memory provided on the CRAY-2 and

planned for the CRAY-3 provides a possible alternative: that of memory-resident data sets. One obvious technique is to generate the integral list in memory for use in subsequent SCF steps. With the use of high symmetries (generating only symmetry-distinct integrals) and pre-screening techniques to eliminate small integral batches, it is possible to perform calculations with 500 basis functions or more in 150 MW or less on the CRAY-2.[28, 92] It is advantageous to avoid storing labels with the integrals by processing the list using the same loop structure as was used to generate them: in the first "SCF iteration" the nonvanishing distinct integrals are computed, stored, and used in the Fock matrix build, and a flag is set in an index list to indicate that a particular batch was computed; in subsequent iterations the same loops are executed, skipping all processing if the batch was not computed and simply retrieving the batch from memory for the Fock matrix build if it was. While there is some overhead associated with the repeated execution of the outer loops of the integral generation code, considerable storage is saved by eliminating labels. As the flag for each batch can be represented by a single bit, the index list requires almost no space. The whole scheme requires little more CPU time than a conventional SCF scheme, but eliminates almost all I/O processing. This approach has also been used to reduce the I/O associated with the perturbed SCF and CASSCF wave function generation in the ABACUS analytic derivative program.[93, 94]

A more drastic approach to the elimination of I/O is not to store large data sets (such as the AO integrals) but to recompute them as required. This is the philosophy behind the "direct SCF" scheme of Almlöf and co-workers.[47, 95] Viewed naively, this appears to represent a trade-off between the CPU time required to generate the integrals and the I/O overhead (and storage requirements) associated with repeatedly retrieving them from secondary storage. However, it is perfectly possible for machines like the X-MP or CRAY-2 to generate the integrals over basis sets of 1500 functions or more in reasonable CPU times (say, on the order of hours); that is, to generate an integral list far larger than the capacity of most supercomputer secondary storage systems. In such a case the question of a trade-off does not even arise, as the conventional approach would not be feasible. Of course, since the integrals must be regenerated in each iteration, it is desirable to minimize the number of iterations and to eliminate as many integrals as possible, as well as using the most efficient scheme possible for computing the integrals. These aspects, together with timings, have been discussed elsewhere,[47, 95, 96] as has an interesting hybrid approach with explicit storage of some integrals.[97]

The extension of this direct approach to MCSCF and CI calculations has also been discussed, but only in purely formal terms without computational implementation.[98] Very recently, however, Saebø and Almlöf[99]

have developed a program for computing the MP2 correlation energy using a direct approach. Such a scheme can be regarded as a first step towards implementing a CI method, as well as generating results that are very useful in themselves. The MP2 energy requires MO integrals of the form $(ia|jb)$, which are most efficiently obtained not by transforming the charge distributions, as described above, but by transforming the first and third indices as the first pair, then the second and fourth. It is obvious that this requires an integral list that is four times longer than the "canonical" list, that is, double the length required for the conventional transformation. In a direct approach to MP2, then, about four times as many integrals must be calculated as in an SCF iteration, so it would be expected that the MP2 energy would require about four times the CPU time of a direct SCF iteration, assuming that integral generation is the dominant step. This is essentially what is observed in practice by Saebø and Almlöf.[99] A similar approach has also been investigated by Head-Gordon *et al.*[100]

4.10. The Influence of Supercomputers on Quantum Chemistry

It is important to conclude this presentation of supercomputer implementations with a discussion on how these algorithms and machines have influenced our approach to quantum chemistry. While it might be thought that performance gains of an order of magnitude or more would simply increase the size of systems considered, their influence is much more profound. For example, the ability to perform full CI calculations in realistic basis sets has provided us with detailed benchmarks for other correlation methods.[67, 68] These have shown that multireference CI wave functions (with some correction for size-consistency effects if eight or more electrons are correlated) reproduce the full CI results to very high accuracy. Since MRCI wave functions are thus an adequate solution to the correlation problem, we may infer that (assuming relativistic effects or Born–Oppenheimer breakdown terms are negligible) any discrepancies between MRCI results and experimental observations are due to inadequacies in the atomic orbital basis. While this had been hypothesized on a number of occasions,[101–103] full CI calculations (being more practical than the use of a complete orbital basis) were required to confirm it. It then becomes worthwhile exploring the possibility of using better basis sets together with MRCI wave functions, and supercomputers provide the necessary computational resources to allow the use of atomic natural orbital basis sets,[51] which have led to almost chemical accuracy (1 kcal/mol) for systems such as N_2,[104] and to even higher accuracy for CH_2 and SiH_2.[105]

Another way in which the power of supercomputers influences quantum chemistry is the speed with which results can be obtained. Even if results of a desired accuracy can be obtained on, say, a VAX-type mini-

computer, the real time required to obtain the results may be too long for the calculation to be useful. It is an important consequence of using a supercomputer that results can be obtained in relatively short times, and this factor will doubtless increase in importance as supercomputer performance increases.

5. Dynamics

5.1. General Observations

While much of the effort in computational quantum chemistry goes into implementing a few, rather well explored methods, this is much less true in scattering calculations. It is generally necessary to consider a wider range of possible methods and their various implementations, and the optimum course is often much more application dependent. We shall review here a greater variety of different methodologies than was the case for quantum chemistry, but there is still no intention to review the entire field; we concentrate on approaches that we have explored and used.

5.2. Classical Dynamics

The simulation of collisions by the quasi-classical trajectory method can be broken down into three steps: the specification of the initial conditions for the trajectories, the integration of the equations of motion to determine the final conditions, and the analysis of the final conditions to extract rate data.[4, 5] The initial coordinates and momenta are determined from quantities that fall into two categories, namely, those which are fixed, like the total energy or initial quantum state, and quantities such as orientations, which are not experimentally resolved. The unresolved quantities must be averaged over, and this necessitates the determination of many different trajectories.

Computationally, the most expensive step is the integration of the trajectories. It is therefore advantageous to consider the savings possible by vectorizing this step. The problem is to determine the coordinates q_i and their conjugate momenta p_i at some time after the collision, given values before the collision. These are determined by solving Hamilton's equations:

$$dp_i/dt = -\partial H/\partial q_i \tag{18}$$

and

$$dq_i/dt = \partial H/\partial p_i \tag{19}$$

Here H is the Hamiltonian and i runs from 1 to the number of degrees of

freedom in the system. Thus once the center-of-mass motion is removed, Hamilton's equations comprise a set of $6N-6$ coupled first-order differential equations, where N is the number of atoms. In the absence of momentum-dependent potentials, the derivatives of the Hamiltonian become $\partial H/\partial q_i = \partial V/\partial q_i$ and $\partial H/\partial p_i = \partial T/\partial p_i$, where V is the potential energy of the system and T is the kinetic energy.

Many different algorithms have been used to solve Hamilton's equations; however, in most methods the time-consuming steps consist of forming the quantities $\mathbf{f}_j$, the vector of time derivatives of the p_i and q_i at intermediate step j, and then using them to predict new coordinates and momenta. For later convenience let $\mathbf{y}_n$ be the vector of p_i and q_i at time step n. The operations for predicting new coordinates and momenta are typically SAXPY-like operations, which can be evaluated reasonably efficiently, as demonstrated in previous sections, provided the vector lengths are great enough. If a single trajectory is being integrated, then the vector lengths would be $6N-6$, or 12 for A + BC collisions and 18 for AB + CD collisions. These lengths are insufficient to give execution rates that approach the ultimate capabilities of vector pipelined machines. Further, the vector lengths involved in the calculation of the gradients of the potential that contributes to $\mathbf{f}_j$ will be even less. Vectorizing a single trajectory is thus not a very efficient use of Cray computers.

It is fortunate that it is possible to do much better by taking advantage of the fact that many trajectories are required. Because the integration of each trajectory is independent of all others, several can be processed simultaneously. That is, new vectors $\mathbf{Y}_n$ and $\mathbf{F}_j$ can be constructed by simply concatenating the vectors $\mathbf{y}_n$ and $\mathbf{f}_j$ from different trajectories; these new longer vectors can then be used in the predictions of new coordinates and momenta. The vector lengths in these steps can thus be made equal to $(6N-6)N_{\text{traj}}$, where N_{traj} is the number of trajectories integrated simultaneously. This means that with little difficulty the asymptotic rates can be reached for the operations that are performed. However, the integration is not the entire calculation, and several other steps need to be considered before predictions of overall execution rate can be made.

We note first that although the integration of each trajectory is independent of the others, the initial conditions are not. This is because the random sampling used to generate the initial conditions relies on pseudorandom number generators, which require knowledge of one or more previous pseudorandom numbers in order to generate the next in the sequence. Thus in general this initialization part of the calculation must be performed in scalar mode. Fortunately, it is of sufficiently small size that the scalar operations do not contribute significantly to the overall run time.

An additional possible complication is that if variable step size algorithms are used to integrate Hamilton's equations, then operations that are

more dependent on the specific trajectory are required to predict the coordinates and momenta. This would disrupt the vector processing discussed above. Consequently it is necessary either to use a fixed step size algorithm, or to modify existing variable step size methods to treat the different trajectories in a more uniform manner. The optimum solution will be problem dependent, and in many cases the simplicity of a fixed step size algorithm will outweigh the advantages of a variable step size algorithm.

The final aspect to consider is the evaluation of the time derivatives $\mathbf{f}_j$, that is, the right-hand sides of (18) and (19). For systems for which Cartesian coordinates are used as the q_i, the derivatives of T are simply mass factors times the p_i and thus are relatively trivial to form compared with the other parts of the calculation. Thus the calculation of dp_i/dt and dq_i/dt, given values of p_i and q_i, is mainly spent evaluating the quantities $\partial V/\partial q_i$. This evaluation can be further broken down into two steps. First of all, it is unlikely that the potential function will be known explicitly in terms of the integration coordinates q_i: typically the interatomic distances might be used to express the potential but Cartesian coordinates for the q_i. Thus it is necessary to first compute the gradients with respect to some set of internal coordinates Q_n, and then to use the chain rule to obtain the final derivatives. Usually the manipulations required for the chain-rule calculations are straightforward and inexpensive compared to the last remaining step, the evaluation of the gradient of the potential. For systems in which a realistic potential is used, this step will usually dominate all other steps. This is simply a manifestation of the complexity of a typical expression for the gradients of the potential compared to the expressions for the integration algorithms.

For relatively simple potentials, it can be advantageous to optimize the various operations taken by the integrator more carefully—see Ref. 106 for an example of how this can be done.

The operations required to generate the gradient of the potential depend on the representation of the potential: typically, mathematical functions like exponential and square root, and various trigonometric functions are required, as well as the usual arithmetic operations. In addition, functions requiring table lookups, such as splines, are sometimes utilized; thus special effort is required to produce an efficient code. However, in contrast to procedures that rely heavily on matrix multiplication, the maximum execution rate that is ultimately achievable is usually well below the theoretical hardware limits. This arises for two reasons. First, the expressions for the gradients often contain common factors, which are the result of single operations: these cannot be chained with other operations and such steps will produce a maximum of a single result per clock period. Second, the possibilities for minimizing memory traffic by unrolling loops or by using several vectors (which remain in the vector registers) for more

than one operation are limited. This is again due to the complexity of the expressions for the gradients and the many different vectors involved. Thus while all of the time-consuming steps for classical dynamics calculations can be vectorized, the vector speedups obtainable are limited because of the relatively inefficient mapping to the hardware.

Perhaps the best way to improve the performance of the construction of the gradients of the potential is simply to minimize the number of evaluations required, that is, to use an efficient variable step size integrator. However, this can cause inefficiencies elsewhere, because of vectorization difficulties, and the various parts of the calculation must be judiciously balanced for overall efficiency. One example where the extra work involved in a variable step size algorithm paid off was in calculations on the recombination of hydrogen atoms.[107] Here one H_2 molecule will have an energy near or above the dissociation limit, and at certain times this energy can be mostly kinetic energy, in which case small time steps will be required, while at other times the energy will be mostly potential energy and the slow velocities would allow larger time steps. Since most of the time will be spent in regions of configuration space where the energy of the diatomic is mostly potential energy, a variable step size algorithm can provide significant savings over a fixed step size algorithm. The variable step size algorithm implemented for this problem was a modification of the Bulirsch–Stoer method.[108] This method has been shown to be an efficient choice for the scalar integration of trajectories,[4] and has the additional advantage that minor modifications allow the vectorization of most of the operations involved in the integration. We only outline the ideas here; full details are given in Ref. 107. The basic philosophy of the method is that to propagate the solution over some time increment, a series of integrations using a low-order method and smaller and smaller step sizes is performed. The results of each individual integration will not necessarily be very accurate, but accurate results can be obtained by extrapolating the results of the different step sizes. Based on the number of integrations required to obtain an accurate extrapolation, the time increment to be used next is predicted, that is, the step size is adjusted. In typical scalar implementations of this method, the number of low-order integrations per time increment is increased until the extrapolation converges within some tolerance. However, a similar scheme would not be very feasible for the integration of several trajectories simultaneously. Thus the method was modified to perform a fixed number of low-order integrations per time increment. Then, based on the errors in the extrapolations using the different numbers of integrations, the results for that time increment are either discarded and the time increment for that trajectory decreased, or the results are saved and the time increment adjusted to reach some error goal. Thus the scalar operations for the variable step size part of the code consist only of error

checking, saving or discarding the results, and adjusting the time increment. These steps amount to such a small fraction of the overall process that this algorithm performs very efficiently on the Cray machines.

As an illustration of the discussion above, we offer the times given in Table 11. Here we show timings using the variable step size Bulirsch–Stoer integrator for two systems. The first system is $F + H_2$ using the simple Muckerman No. 5 LEPS potential.[109] This system was chosen to give an idea of the limit of a three-body system with a simple potential. The second system is $H_2 + H_2$ using the accurate potential from Ref. 107, which is based on extensive *ab initio* electronic structure calculations, and represents the extreme case of a complicated potential for a system with many degrees of freedom. Both calculations use interatomic distances as the internal coordinates Q_n with respect to which the potential gradients are directly calculated, and Cartesian coordinates having the center of mass stationary at the origin as integration coordinates.[107] The times are normalized so that the integration time is 1 unit on the X-MP/48, and the MFLOPS rates are determined from the hardware performance monitor on the X-MP/48 and by scaling by the appropriate times for the other machines. The number of simultaneously integrated trajectories (N_{traj}) was 500.

First, consider the $F + H_2$ system. Even when using the very simple potential employed here, a considerable fraction of the total time is spent evaluating the gradients of the potential with respect to the integration coordinates. The amount of time spent in the integration routine amounts to only 36%–51%, depending on the machine. Thus in spite of the modest execution rate of the variable step size part of the code, a reasonable overall rate is achieved. Of the time spent getting the gradients, about 20%–27% is spent calculating the interatomic distances and transforming the gradients to the integration coordinates. Consider now the $H_2 + H_2$ system. Here the complexity of the potential is obvious—94% of the time is spent evaluating the gradients of the potential with respect to the internal coordinates in spite of the fast execution rates of 100–190 MFLOPS. Here since only 3% of the time is spent in the integrator, considerable flexibility is available in optimizing the variable step part of the code by introducing more scalar operations, without significantly degrading performance. The relative CPU time per integration step is about a factor of 20 greater for $H_2 + H_2$ than for $F + H_2$.

The overall execution rates for either of the two potentials on the various machines range from 100 to 190 MFLOPS, with the Y-MP at the top of this range and all the other machines clustered at the bottom. These rates are well below the theoretical hardware limits, which is probably a reflection of the limited optimization ability of current compilers when faced with the complicated expressions present in these codes.

At this juncture, we discuss other resources required by the classical

Table 11. Execution Times and Rates for Classical Trajectories

	Fraction of total time				MFLOPS			
	X-MP/48[a]	Y-MP/832[b]	CRAY-2[c]	CRAY-2*[c]	X-MP/48	Y-MP/832	CRAY-2	CRAY-2*
				$F + H_2$				
Gradients of potential	0.36	0.39	0.29	0.31	130	190	160	180
Build Q_n and chain rule	0.27	0.24	0.20	0.20	120	210	160	190
Subtotal	0.63	0.64	0.49	0.51	126	200	160	184
Integrator	0.37	0.36	0.51	0.49	55	93	40	50
Overall					100	160	99	118
				$H_2 + H_2$				
Gradients of potential	0.94	0.94	0.94	0.94	110	190	100	120
Build Q_n and chain rule	0.03	0.03	0.02	0.03	140	210	140	160
Subtotal	0.97	0.97	0.96	0.97	111	190	101	121
Integrator	0.03	0.03	0.04	0.03	60	110	50	50
Overall					110	190	98	119

[a] Using CFT compiler. The CFT77 compiler produces bad code for this problem.
[b] Using CFT77 compiler.
[c] Using CFT77 compiler. Because of the presence of some features of FORTRAN 77 that are only available under CFT77, we do not use CFT2.

dynamics calculations, namely, disk and memory. As a rule, the variables required for the various operations required to propagate a trajectory are of sufficiently small number that they all can be held in memory, thus very few I/O operations are required during the integration of a trajectory. An exception to this is if the coordinates and momenta along the trajectories are desired for analysis purposes, such as plotting. Then it will be necessary to perform a certain amount of I/O in order to save these quantities for later use. However, the majority of trajectories will not be so analyzed, and thus it is only necessary to save the initial and final conditions. The memory allocated by our program has not been minimized, mainly because we have not encountered problems obtaining the space desired. Much of the space is taken up by temporary vectors, which a more sophisticated compiler could allocate dynamically, thus reducing the overall space requirements. The present code used for $H_2 + H_2$ requires $720 \times N_{traj}$ words for the integration part of the code and $589 \times N_{traj}$ words for the potential gradient part of the code. Thus when using $N_{traj} = 500$, which is our usual production value, the code requires less than one million words of memory.

This section will be closed with some ideas on how the timings for trajectories could be improved. Since most of the time is spent on evaluating the gradients of the potential, we will concentrate on this aspect of the problem. With the recent availability of *ab initio* gradient methods, it would seem advantageous to directly fit the gradients of the potential. Then one would be evaluating different functions for the various gradients rather than differentiating a single function. Although this procedure has many conceptual merits, it has several possible problems that limit its usefulness. First of all, the fitted gradients will probably not all integrate to the same function, that is, the potential will be nonconservative[110] and so the trajectories will not conserve energy. However, in practice this may not be a significant problem. Another problem is that often both the potential and the gradients are required. For instance, it is often desirable to calibrate classical methods by comparing to quantum mechanical solutions of the problem, and quantum mechanical dynamics methods require the potential itself rather than the gradients. In addition, hybrid dynamical methods exist[111] which treat some degrees of freedom using classical mechanics and some using quantum mechanics. Another facet of this reflects the large amount of labor usually involved in constructing a fit to the potential: it is not uncommon for several different studies using different methods to be performed using a potential once it has been fitted. It is thus less restrictive if the potential is given rather than the gradients. Finally, it may simply not be as efficient to evaluate separate fits to the gradients as it is to explicitly differentiate a function. This is because there may be many common expressions in the differentiation formulas. For example, if we make the assumption that the gradients will be fitted by a function of similar com-

plexity as the potential, then a comparison of the number of gradients times the time to evaluate the potential to the time to evaluate the potential gradients will give some idea of the efficiency of the method. For the simple $F + H_2$ potential used in the timings above, the time to generate the potential and three gradients is only 1.3 times as long as just generating the potential, and for the $H_2 + H_2$ potential of Ref. 107, the time to generate the potential and six gradients is only 2.3 times the time for just generating the potential alone. Thus for these cases, it is more efficient to deal with the potential and then explicitly differentiate it.

5.3. Quantum Dynamics

In the differential equation approach to quantum mechanical dynamics calculations of nonreactive molecular collisions, the equations to be solved are[112]

$$\frac{d^2}{dr^2}\mathsf{f}(r) - \mathsf{D}(r)\,\mathsf{f}(r) = 0 \tag{20}$$

where r is the distance between the centers of mass of the target and projectile, the f are the unknown radial functions, and D is the coupling matrix. The matrix elements of the coupling matrix are given by

$$D_{nn'} = \frac{2\mu}{\hbar^2}\langle n|\, V^{\text{int}}\,|n'\rangle + \delta_{nn'}[-k_n^2 + l_n(l_n+1)/r^2] \tag{21}$$

where $\langle n|\, V^{\text{int}}\,|n'\rangle$ is the matrix element of the interaction potential between the basis functions labeled by the indices n and n', μ is the reduced mass for the collision partners, k_n^2 is the square of the wave vector for channel n:

$$k_n^2 = 2\mu(E - \varepsilon_n)/\hbar^2 \tag{22}$$

E is the total energy, ε_n is the internal energy for channel n, and l_n is the orbital angular momentum quantum number for relative translational motion for channel n. The basis functions labeled by n describe all degrees of freedom except r and each value of n defines what is known as a channel. The interaction potential is defined as that part of the potential that goes to zero as r goes to infinity; thus it does not include the binding potential of the collision fragments.

The boundary conditions for the unknown functions are

$$f_{nn'}(0) = 0 \tag{23a}$$

and as r goes to ∞,

$$f_{nn'} \sim \begin{cases} (2ik_n)^{-1/2}\left\{\delta_{nn'}\exp\left[-i\left(k_n r - l_n\frac{\pi}{2}\right)\right] - S_{nn'}\exp\left[i\left(k_n r - l_n\frac{\pi}{2}\right)\right]\right\}, & k_n^2 > 0 \\ i(2|k_n|)^{-1/2}\{\delta_{nn'}\exp[|k_n|r] - \tilde{S}_{nn'}\exp[-|k_n|r]\}, & k_n^2 < 0 \end{cases} \tag{23b}$$

From the matrix elements of the scattering matrix, $S_{nn'}$, experimental observables can be calculated using standard formulas.[113–115]

To determine the scattering matrix, we convert the problem of determining the f from a boundary value problem to an initial value problem by determining a linearly independent set of solutions of (20), called $\tilde{\mathsf{f}}$, which are regular at the origin but have arbitrary slopes there.[116] In practice it is not usually necessary to start at the origin but rather at any larger distance where the hard core of the potential is sufficiently repulsive so that the radial wave functions are negligibly different from zero. At large r these functions behave as

$$\tilde{f}_{nn'} \sim \begin{cases} P_{nn'}\sin\left(k_n r - l_n\frac{\pi}{2}\right) + Q_{nn'}\cos\left(k_n r - l_n\frac{\pi}{2}\right), & k_n^2 > 0 \\ \tilde{P}_{nn'}\exp[|k_n|r] + \tilde{Q}_{nn'}\exp[-|k_n|r], & k_n^2 < 0 \end{cases} \tag{24}$$

with P, $\tilde{\mathsf{P}}$, Q, and $\tilde{\mathsf{Q}}$ determined from $\tilde{\mathsf{f}}$ at two points or the logarithmic derivative at one point. From P and Q, the scattering matrix can be easily determined.

The number of algorithms for solving the close-coupling equations (20) is quite large,[117–121] and the optimum algorithm is case dependent. The number of coupled equations, the dimension of D in (20), will be called N and can range from 1 for central potential problems to over a thousand for anisotropic AB + CD collisions;[122] thus the vectorization strategies will depend on the size of the problem. In contrast to problems in classical mechanics, the integration of (20) for a single channel ($N = 1$) can be vectorized relatively effectively. This is because the interaction potential depends only on the parameter r, which can be made to take on known values while in classical problems the potential depends on the unknown functions that are being determined. Thus the steps required for integrating the close-coupling equations for a single channel typically would be to determine $D_{11}(r)$, to determine temporary quantities depending on $D_{11}(r)$ for single values of r, and finally to recursively assemble the solution. The first two steps are completely vectorizable, with vector components corresponding to different values of r, while the final step is not;

however, if properly coded this final step will not involve many operations per r value and hence will not be the rate-determining step. For example, in Ref. 123, scattering calculations were carried out at complex energy searching for poles in the scattering matrix. This required calculations at many different energies. The algorithm used was a modification of the R-matrix propagation algorithm,[124–126] where the equations defining the modified method are

$$\tilde{R}_4^{(1)} = 0 \tag{25}$$

$$\tilde{R}_4^{(i)} = t^{(i)}/(\tilde{R}_4^{(i-1)} + q^{(i)}) \tag{26}$$

$$f_{11}/(df_{11}/dr|_{r=r_i+h_i/2}) = -\tilde{R}_4^{(i)} + r_1^{(i)} \tag{27}$$

$$t^{(i)} = -[P_3^{(i)}]^{-2} \tag{28}$$

$$q^{(i)} = r_1^{(i)} + r_1^{(i-1)} \tag{29}$$

$$r_1^{(i)} = P_1^{(i)}[P_3^{(i)}]^{-1} \tag{30}$$

$$P_1^{(i)} = \cosh(\lambda^{(i)} h^{(i)}) \tag{31}$$

$$P_3^{(i)} = \lambda^{(i)} \sinh(\lambda^{(i)} h^{(i)}) \tag{32}$$

and

$$\lambda^{(i)2} = D_{11}(r_i) \tag{33}$$

In the above equations, $h^{(i)}$ is the width of sector i and r_i is the center of sector i. Equations (25) and (27) need be evaluated only once per calculation, equation (26) is the recursive step, and the other equations are vectorizable with vector lengths equal to the number of sectors, which is typically on the order of hundreds. Thus significant vector speedups are obtainable. It should be noted that as in the case of classical dynamics, the vectorized steps here do not map as well to the Cray architecture as operations such as matrix multiplication do, so that extremely high execution rates are not possible.

For calculations with large N, it will be more efficient to vectorize the operations required for the individual integration steps. The majority of the time will be spent on standard matrix manipulations on matrices of order $N \times N$. For large N, these operations will dominate the calculation because they scale as N^3. Since (depending on the algorithm) various numbers of matrix multiplications, matrix inversions, linear equation solutions, and matrix diagonalizations will be required per integration step, the choice of a particular algorithm will be based on the relative work of the various operations, that is, the coefficient multiplying N^3, and also on the execution rate of the operations, which introduces another coefficient multiplying the operation count. We now consider these last points in detail.

In Table 12, we quote estimates of the asymptotic execution rates and matrix order required to achieve half the asymptotic rate for several matrix operations. The operation count for large enough N will scale as $C_x N^3$, with C_x a coefficient depending on the operation. For matrix multiplication we take $C_{M\times M}=2$; for general linear equation solution we use $C_{GLS}=\frac{8}{3}$ ($2N^3/3$ for LU decomposition, and $2N^2$ per right-hand side for forward elimination and back substitution); for symmetric linear equations we use half this value: $C_{GLS}=\frac{4}{3}$. Finally for real symmetric matrix diagonalization, we estimate $C_{RS}=10$. These operation counts are based on the discussions of Ref. 127. The data in the table were determined by averaging the results obtained by fitting the CPU times and MFLOPS rates with matrices of order 63, 127, 255, and 511 to $C_x N^2(t_s+Nt_\infty)$ or $10^{-6}/(t_\infty+t_s/N)$ by least squares. These particular orders were chosen to be close to multiples of 64 to make most efficient use of a Cray computer's 64-element vector registers while at the same time avoiding multiples of 2, which can cause catastrophic bank conflicts as discussed in Section 3 above. For example, the execution rate for 512 is a factor of 20 less than for 511 on the CRAY-2 when using the general linear equation routine LUSOLV. We should also point out that by fitting only these above array dimensions we obtain different $n_{1/2}$ values from those in Section 3 above. In particular, the fit includes only array dimensions considerably larger than the true $n_{1/2}$ values for matrix multiplication, and so no $n_{1/2}$ results are given for this case.

Table 12. Characterization of the Execution Rates for Matrix Operations

	Routine						
Machine	MXM[a]	RS[b]	LUSOLV[c]	LUSOLV[d]	MINV	SGEF A/SL[e]	SSPFA/SL[f]
X-MP/48	200[g]	170	190	210	73	130	73
	—[h]	150	110	120	0	200	470
Y-MP/832	290	240	—	300	120	250	120
	—	100	—	110	7	150	470
CRAY-2	310	120	170	240	340	120	43
	—	25	160	290	80	140	330
CRAY-2*	370	140	210	200	360	140	50
	—	30	160	150	60	140	280

[a] Matrix multiply routine from SCILIB.
[b] EISPACK[57] routine for real symmetric eigenvalues and eigenvectors from SCILIB.
[c] FORTRAN program for general linear equation[129,130] compiled using CFT2.
[d] FORTRAN program for general linear equation solution compiled using CFT77.
[e] LINPACK[131] program for general linear equation solution from SCILIB.
[f] LINPACK program for symmetric linear equation solution from SCILIB.
[g] Upper entry: estimate of asymptotic execution rate in MFLOPS; lower entry: fitted estimate (see text) of matrix order required to achieve half of the asymptotic rate.
[h] Matrices used are too large to allow reliable estimate of this quantity (see text).

For matrix multiplication, we give results for the Cray routine MXM. For diagonalization, we only quote results for the EISPACK[57] routine RS, which we have found most convenient and reliable for our scattering calculations, although other routines may be more efficient.[128] In the following comparison we will emphasize asymptotic execution rates (r_∞), but for finite matrices the $n_{1/2}$ values are also important parameters.

For linear equations, we give results for four different routines. The first, called LUSOLV, is a FORTRAN code utilizing loops unrolled to a depth of 16.[14, 129, 130] The next is the Cray library routine MINV.[22] The last two are LINPACK[131] routines, one for a general matrix and one for symmetric matrices. Comparing the four methods for linear equation solution, we see that it is never advantageous from a CPU time point of view to use the specialized symmetric matrix routine, for its execution rate is always much more than a factor of 2 less than the other routines. While it uses less memory, since only a triangle of the matrix needs be stored, the execution rates are so poor that this is unlikely to be of sufficient motivation to use it. On the X-MP/48, the execution rates of the general LINPACK routine and the library routine MINV are similar, but the LUSOLV program is about a factor of 2 more efficient than those routines. On the Y-MP the situation is similar, with the LUSOLV program being the fastest, followed by the general LINPACK routine, and then MINV, with relative ratios 1:1.2:2.5. On either CRAY-2 machine the situation changes, and the Cray library routine MINV is almost a factor of 2 faster than LUSOLV, and more than a factor of 2.5 times faster than the general LINPACK routine. Thus the most efficient routine to use on the X-MP or the Y-MP is the FORTRAN program LUSOLV, whereas on the CRAY-2 the most efficient routine is CRI's MINV.

The relative efficiencies of the various operations can be estimated by comparing the r_∞ values (see also Section 3). Since matrix multiplication is usually the most efficient operation, it is convenient to introduce the ratios $\mathscr{R}_{RS}$ and $\mathscr{R}_{LE}$ which measure the asymptotic rates compared to matrix multiplication for diagonalization and the best method for linear equation solution. On the X-MP and Y-MP we have $\mathscr{R}_{RS} = 1.2$ and $\mathscr{R}_{LE} = 1$, while on the CRAY-2 these ratios are 2.6 and 0.91–1.0, depending on the memory speed. On the X-MP and Y-MP, therefore, these various matrix operations are approximately equally efficient, while on the CRAY-2 the diagonalization code is much less efficient than the others. However, on all Cray machines, the $n_{1/2}$ values are much shorter for matrix multiplication than for the other operations, so for small to moderate matrices, the diagonalization and linear equation solution operations will be relatively less efficient than the above discussion indicates. We should again point out that these timings (especially CRAY-2 values) are subject to about 10% fluctuation, depending on system loads.

Based on the timings reported above, it seems desirable to maximize the number of matrix multiplications and minimize the other operations in the overall scheme. One algorithm that does this, at least in principle, is the DeVogelaere method,[132] which only requires matrix multiplications. However, in practice, two aspects diminish the attractiveness of this method. First of all, the r step size is limited by the oscillation of the wave functions, that is, a certain number of integration steps will be required per De Broglie wavelength. Thus for long-range potentials or high-energy collisions, an extremely large number of integration steps will be required. In contrast, a number of algorithms exist where the integration step size is controlled by the change in the interaction potential and thus large steps can be taken where the potential is slowly varying. The step sizes required for accurate results for these methods are only weakly dependent on total energy also.

The second difficulty with the DeVogelaere method is one common to all initial value methods that explicitly determine the radial functions $\mathbf{f}$. At small r, all channels will be classically inaccessible, and thus the functions will be growing exponentially. If the classically forbidden region is too large, then the component of the functions that has the largest exponential growth will become sufficiently larger than all other components that (to working precision) the linear independence of the initial conditions will be lost. The determination of the scattering matrix will then not be possible.[132] This can be controlled to a certain extent by periodically reorthogonalizing the columns of $\mathbf{f}$,[133] but this stabilization procedure introduces extra operations and a small integration step may be required to limit the exponential growth per step. In practice, for molecular collisions where it is necessary to include channels that are classically inaccessible for all r, this difficulty can be a severe problem.

An alternative solution is to devise an algorithm that does not explicitly determine the $\mathbf{f}$ but rather some other quantity, which will be inherently stable. For example, the logarithmic derivative matrix $\mathbf{f}^{-1}\, d\mathbf{f}/dr$ will not suffer from stability problems.

Because of these numerical difficulties, we have found the R-matrix propagation algorithm to be very attractive.[124–126] In this method, the integration step size is usually limited by the r derivative of the coupling matrix D, so large step sizes are possible in the asymptotic region where the interaction potential is slowly varying, which is particularly advantageous for long-range potentials. In addition, the negative of the inverse of the logarithmic derivative matrix is propagated, which avoids all stability problems. A final advantage is that a large fraction of the work required is independent of total energy E so that effort can be saved by performing calculations for several energies. A disadvantage is that operations other than matrix multiplications are required.

The operations involved per integration step for this algorithm are as follows. The energy-independent steps are to construct the coupling matrix D (which is real and symmetric) in sector i, to diagonalize it to determine the local adiabatic eigenvectors T^i and eigenvalues $[\lambda^{(i)}]^2$, and then to form the overlap matrix

$$\mathscr{T}(i-1, i) = [\mathsf{T}^{(i-1)}]^T \mathsf{T}^{(i)} \tag{34}$$

For calculations at subsequent energies, it is only necessary to save the local adiabatic eigenvalues and overlap matrices. Thus for large N, the relative time for the energy independent step will be $N^3(2+10\mathscr{R}_{RS})$. Since the total energy enters in equation (21) only as a multiple of the unit matrix, the local adiabatic eigenvectors are independent of E and the eigenvalues are shifted by a constant amount if E changes. Then at each energy it is necessary to form

$$\mathsf{R}_4^{(i)} = \mathsf{r}_4^{(i)} - [\mathsf{P}_3^{(i)}]^{-1} \{\mathsf{R}_4^{(i-1)} \mathscr{T}(i-1, i) + \mathscr{T}(i-1, i) \mathsf{r}_4^{(i)}\}^{-1} \mathscr{T}(i-1, i) [\mathsf{P}_3^{(i)}]^{-1} \tag{35}$$

where

$$\mathsf{R}_4^{(i)} = -\mathsf{f}[d\mathsf{f}/dr]^{-1}|_{r=r_i+h_i/2} \tag{36}$$

$$\mathsf{r}_4^{(i)} = [\mathsf{P}_3^{(i)}]^{-1} \mathsf{P}_1^{(i)} \tag{37}$$

$$[\mathsf{P}_1^{(i)}]_{nn'} = \delta_{nn'} \cosh(\lambda_n^{(i)} h_i) \tag{38}$$

and

$$[\mathsf{P}_3^{(i)}]_{nn'} = \delta_{nn'} \lambda_n^{(i)} \sinh(\lambda_n^{(i)} h_i) \tag{39}$$

Since $\mathsf{P}_1^{(i)}$, $\mathsf{P}_3^{(i)}$, and $\mathsf{r}_4^{(i)}$ are diagonal, the time-consuming operations required for (35) are one matrix multiplication and one linear equation solution, and so we see that the relative time for the energy-dependent step is $N^3(2+8/3\mathscr{R}_{LE})$. If we consider large-scale calculations on the CRAY-2, where we take $\mathscr{R}_{RS}$ as 2.6 and $\mathscr{R}_{LE}$ as 1, then we see that the ratio of the energy-independent time to the energy-dependent time is six; thus it is advantageous to perform calculations at many energies—performing calculations at seven energies would require only about twice as much CPU time as one energy. (Note that an earlier version of our code[130, 134] used a different sequence of matrix operations that was less efficient both in time and memory usage.) This assumes that the time for the evaluation of the matrix elements of the interaction potential required for (21) is negligible. This is not always the case, even though the work to construct this matrix should scale as N^2 for large enough N. If the interaction potential time is not negligible, then the ratio of the energy-dependent times to the energy-independent times will be even greater. Strategies for optimizing the interaction potential matrix evaluation will be discussed below.

The storage requirements for the *R*-matrix propagation algorithm are relatively small, considering the amount of memory available on the CRAY-2. The matrices that need to be stored are of order $N \times N$, and the number required is not large. For a single energy calculation, our present code requires seven matrices of this size, excluding the data required to form the interaction potential matrix. Usually we keep these matrices in memory, but some calculations carried out on a CRAY-1 used a version of the code that kept only two matrices in core and the remainder on disk. For the values of N we used in those calculations (440 and 530), the I/O requirements of this strategy did not significantly degrade the performance of the method. This would be especially true on a machine with a large SSD. Other quantities that need to be stored are those required to evaluate the potential coupling matrix. The requirements here may or may not be considerable, depending on the potential used. In our large-scale HF + HF calculations, using a complicated potential, our code required approximately an additional $40N^2$ words of storage for angular integrals and pointers (see below for a description of the construction of the potential matrix). Fortunately, these data are accessed sequentially, thus comparatively little is lost by storing them on secondary storage like disk or an SSD.

If more than one energy is to be used, there are two storage choices. The first choice is to finish the calculation at the first energy before performing any part of the calculation for other energies. This requires that an additional N_{step} matrices be saved (the sector overlap matrices), where N_{step} is the number of integration sectors used. The second choice is to perform the calculations for all energies at sector i before going on to the next sector. This requires $8 + N_E$ matrices, where N_E is the number of energies used.[134] Thus since N_{step} is usually on the order of hundreds and N_E on the order of tens, it is usually more efficient to make the second choice.

We now turn to the question of evaluating the interaction potential matrix. It is usually most efficient to transform to the body-fixed frame of reference to evaluate the interaction potential matrix. In this frame the kinetic energy part of D is no longer diagonal, but the coupling is very simple.[135] It is necessary to back-transform to the laboratory frame only when the asymptotic analysis to determine the scattering matrix is carried out. For A + BC collisions we need to compute the body-frame matrix elements

$$\begin{aligned}\langle vj\Omega JMP|\, V^{\text{int}} \,|v'j'\Omega JMP\rangle \\ = 2\pi \int dR \int d(\cos\chi)\, Y^*_{j\Omega}(\chi, 0)\, \Theta^*_{vj}(R)\, V^{\text{int}}(r, R, \chi) \\ \times Y_{j'\Omega}(\chi, 0)\, \Theta_{v'j'}(R) \end{aligned} \tag{40}$$

where v and v' are vibrational quantum numbers, j and j' the diatomic rotational angular momentum quantum numbers, J the total angular momentum, M the projection of J on the laboratory-frame z axis, Ω the projection of j, j', and J on the body-frame z axis, P the parity, R the diatomic bond length, r the distance between A and the center of mass of BC, χ the angle between the vectors from A to the center of mass of BC and C to the center of mass of BC, $Y_{j\Omega}$ a spherical harmonic, and Θ_{vj} a vibrational function. To arrive at (40) we have averaged over the Euler angles rotating an arbitrary laboratory-fixed coordinate system to the body fixed coordinate system: this produces a Kronecker delta for Ω, J, M, and P, and the factor 2π. The traditional way to proceed is to expand the cos χ dependence of the interaction potential in terms of Legendre polynomials and then perform the angular integrals analytically. The R integral is then performed most efficiently using an optimized quadrature[136] so that the matrix elements become

$$\langle vj\Omega JMP|\ V^{\text{int}}\ |v'j'\Omega JMP\rangle = \sum_{i,\lambda} w_{vjv'j',i} v_\lambda(r, R_i) f^{\Omega}_{jj'\lambda} \tag{41}$$

where $w_{vjv'j',i}$ is a vibrational quadrature weight, v_λ is a potential expansion coefficient, and $f^{\Omega}_{jj'\lambda}$ is an angular integral (which is independent of M, J, and P). In general, it will also be necessary to generate the potential expansion coefficients, and this is most straightforwardly done by projection:

$$v_\lambda = \frac{2\lambda+1}{2} \int d(\cos\chi)\, P_\lambda(\cos\chi)\, V^{\text{int}}(r, R, \chi) \tag{42}$$

The quadrature approximation to (42) is most efficiently evaluated as a matrix–vector product. Usually if the potential is given explicitly in terms of a Legendre expansion there are only a few terms in the λ sum; however, if it is necessary to converge the angular expansion of a more general potential, several tens of terms are typically required. The difficulty of efficiently evaluating (41) arises for two reasons. First, the angular integrals are zero unless the triangle rule for j, j', and λ is satisfied; thus many integrals are zero and it is advantageous to exploit that. Second, the way in which two pieces of information depending only on a subset of the quantum numbers are combined together complicates efficient evaluation.

We now consider three schemes to evaluate (40). In scheme A we evaluate (42) by matrix–vector product and then form

$$\tilde{v}_{vjv'j',\lambda}(r) = \sum_i w_{vjv'j',i} v_\lambda(r, R_i) \tag{43}$$

This is performed as a matrix multiplication with $vjv'j'$ labeling rows and

λ columns of the result. The inner index i is the number of points in the vibrational quadratures and usually is on the order of 10. Finally a sparse SAXPY is performed for each value of λ to generate the matrix element. In scheme B the procedures of A are modified by forming the intermediate product

$$A^{\Omega}_{jj'}(r, R_i) = \sum_{\lambda} v_{\lambda}(r, R_i) f^{\Omega}_{jj'\lambda} \tag{44}$$

which is a sparse SAXPY followed by

$$\langle vj\Omega JMP | V^{\text{int}} | v'j'\Omega JMP \rangle = \sum_{i} w_{vjv'j',i} A^{\Omega}_{jj'}(r, R_i) \tag{45}$$

which is a vector plus vector times vector using scattered data. Finally in scheme C we dispense with the expansion of the potential and directly* form the $A^{\Omega}_{jj'}(r, R_i)$ via

$$A^{\Omega}_{jj'} = \sum_{n} C^{\Omega}_{jn} \tilde{C}^{\Omega}_{nj'}(r, R_i) \tag{46}$$

where

$$\tilde{C}^{\Omega}_{nj} = C^{\Omega}_{jn} V^{\text{int}}(r, R_i, \cos \chi_n) \tag{47}$$

and

$$C^{\Omega}_{jn} = (2\pi \tilde{w}_n)^{1/2} Y_{j\Omega}(\chi_n, 0) \tag{48}$$

$\tilde{w}_n$ a Gauss–Legendre quadrature weight. Equation (46) is evaluated as a matrix multiplication. The number of points in the angular quadrature is on the order of 10.

The three schemes have various advantages and disadvantages. A drawback of schemes A and B is that it is necessary to form the angular integrals $f^{\Omega}_{jj'\lambda}$, while in scheme C it is necessary to form the C^{Ω}_{jn}, which is an easier task and can be done efficiently in vector mode. However, these steps are independent of r and R and so should not be of significant overall cost. In scheme A the vibrational integral is computed very efficiently, but too much work is performed: many of the $\tilde{v}_{vjv'j',\lambda}$ will not be required since many of the $f^{\Omega}_{jj'\lambda}$ are zero. Schemes B and C perform no more work than is required for the vibrational integrals, but (because of the scattering of data because a given $vjv'j'$ pair contributes to more than one Ω) this work will not be performed as efficiently in scheme A.

Turning now to the angular part of the different methods, scheme A

* We are grateful to J. N. Murrell for suggesting this option.

only performs the sparse SAXPYs once per λ, while scheme B performs them once per λ per vibrational quadrature point. Scheme C also needs to repeat the angular integrals for each vibrational quadrature point. However, if there were only one vibrational quadrature point, scheme A would perform more work because the total number of angular integrals (zero and nonzero) per λ is proportional to the square of the number of vj states, whereas for schemes B and C the number is proportional to the square of the number of j states. Thus the relative efficiencies of schemes A and B will depend on particular circumstances, with a larger number of v states coupled with a smaller number of vibrational quadrature points favoring scheme B and a small number of v states using many vibrational quadrature points favoring scheme A. In our experience, usually the efficiency performing the vibrational integrals is the most important factor; thus scheme A is more efficient than scheme B. This is another example of how minimizing the operation count does not necessarily lead to the most efficient algorithm on vector computers.

In most cases, scheme B will be preferred over scheme C because the matrix multiplication to produce the angular integrals takes longer than the sparse SAXPYs. However, in special cases scheme C also has one potentially important advantage: it may be possible to greatly reduce the number of vj states used in the wave function expansion by using some sort of adiabatic rotational states rather than spherical harmonics.[137] That is, a few new functions $\tilde{Y}_{n\Omega}$, which are linear combinations of many of the $Y_{j\Omega}$, could be defined and used instead. This may allow otherwise intractable problems to be solved. A single left transformation of the C_{jn}^{Ω} is all that is required for scheme C, while the other schemes require similarity transformations for each value of λ to produce new angular integrals, which now in general are not sparse, so the sparse SAXPYs are replaced with full SAXPYs. This will increase both the memory and the operation count for these schemes. The transformation times will be important because the transformations depend on r and hence must be carried out many times.

Turning now to the case for AB + CD collisions, the matrix element to be found is[138] $\langle v_1 v_2 j_1 j_2 j_{12} \Omega J M P | V^{\text{int}} | v_1' v_2' j_1' j_2' j_{12}' \Omega J M P \rangle$, where v_i is the vibrational quantum number of molecule i, j_i is the rotational quantum number of molecule i, j_{12} is the quantum number of the vector sum of j_1 and j_2, J is the total angular momentum quantum number, M is its projection on the laboratory frame z axis, Ω is the projection of j_{12}, j_{12}', and J on the body fixed z axis, and P is the parity. The procedure here is very similar to the A + BC case, except that the manipulations are more complicated. Expanding the angular dependence of the potential in terms of combinations of spherical harmonics now involves three angles and three indices; thus the number of terms required is approximately the cube of

that required for the atom–diatom case.[139] For anisotropic potentials, close to a thousand terms can be required to converge the angular representation of the potential.[140] Thus although one may be tempted to ignore the presence of zero angular integrals in the A + BC case to simplify vectorization, for the AB + CD case the number of integrals is so large that it is imperative that the zero integrals be identified and not stored. The aggregate number of angular quadrature points can also be in the thousands. The situation for the vibrational integrals is not quite as bad, because there are only two vibrational coordinates and so the number of quadrature points will be only the square of the A + BC case, say, 50 to 100 if optimized quadratures[136] are used. The three different schemes described above for performing the integrals can be applied here with little modification with the exception of scheme C, which for Ω greater than zero must be modified to include contributions from two numerical integrals—that is, two matrix multiplications are required, because of the more complicated coupling of the angular momentum vectors. Thus C^{Ω}_{jn} of (48) is replaced with the quantity $C^{\Omega x}_{j_1 j_2 j_{12} n}$ given by

$$C^{\Omega x}_{j_1 j_2 j_{12} n} = 2[\pi \tilde{w}_{n_1} \tilde{w}_{n_2} \bar{w}_{n_3}]^{1/2} \sum_{m_1 m_2} (j_1 m_1 j_2 m_2 | j_1 j_2 j_{12} \Omega)\, N_{j_1|m_1|} (-1)^{(m_1 + |m_1|)/2} \times N_{j_2|m_2|} (-1)^{(m_2 + |m_2|)/2}\, P^{|m_1|}_{j_1}(\cos \chi_{n_1}) \times P^{|m_2|}_{j_2}(\cos \chi_{n_2})\, x[(\phi_{n_3}/2)(m_2 - m_1)] \tag{49}$$

where x is either sin or cos, n is a composite index denoting the three quadrature indices n_1, n_2, and n_3, $\tilde{w}_n$ and $\bar{w}_n$ the quadrature weights for the different angular integrations, $(...|...)$ is a Clebsch–Gordan coefficient, N_{jm} the (positive) normalization factor for the spherical harmonic Y_{jm}, P^m_j is an associated Legendre function, χ_1 the center of mass BC to center of mass AB to B angle, χ_2 the inclination angle for the second molecule, and ϕ is the dihedral angle. If Ω is zero, then only the functions with $x = \cos$ are required. For Ω greater than zero, the two matrix multiplications are for cosine terms times cosine terms and sine terms times sine terms. The weighting of the various schemes changes somewhat for this case because of the relative complexity of the analytic calculation of the angular integrals required for schemes A and B. Although these integrals are still independent of r, their calculation can be time-consuming enough that they can consume a significant fraction of the computational resources required for a scattering calculation. The analytic formulas involve 3-*j*, 6-*j*, and 9-*j* symbols, which are generated mostly in scalar mode. In contrast, scheme C, which only requires the functions of (49), involves only a Clebsch–Gordan coefficient (which is a rephased and normalized 3-*j* symbol); thus the time to calculate the functions for scheme C will be much less than the

time to form the angular integrals required for the other two schemes. In addition, the efficient implementation of adiabatic rotational states possible with scheme C makes it rather attractive.

An alternative scheme to reduce the operations required to evaluate the D matrix is to employ some form of discrete variable representation.[141] Here a transformation is made so that the potential coupling is diagonal with elements equal to the potential at grid points. The kinetic energy is not diagonal in this basis. So far this method has been applied only to rotational-like degrees of freedom for three-atom systems. Another way to reduce the effort to evaluate the potential matrix is to write the interaction potential in a special form whereby much of the angular and vibrational integrations need be done only once.[122]

We also reiterate that the potential can be reused for calculations at more than one energy, and if the body-frame matrix elements are used as discussed above, they can be used for more than one value of the total angular momentum J. Finally it may be possible to achieve speedups by evaluating the potential matrix using a coarser grid than is used to integrate (20) and then interpolating the result. Thus if one is willing to perform extensive enough calculations (many energies, many J), the evaluation of the potential matrix will be an insignificant part of the overall calculation, and the overall execution rate is determined by the rate of the matrix manipulations in the algorithm used for solving (20).

We now turn to some new algorithms and methods that have been made viable by the availability of the large CRAY-2 memory. In contrast to the quantum mechanical methods discussed so far, which propagate a solution along r, and hence need only store the relevant data for one value of r, the methods we mention now are more global in nature and require data for all values of r. The first method we discuss is the finite difference boundary value method (FDBVM).[142, 143] Here the close-coupling equations are solved not by specifying two sets of boundary conditions at small r, but rather by specifying a boundary condition at both small and large r. This can be done in many ways, and the easiest is to set up a grid of r consisting of the points $r_1, r_2, ..., r_{N_{\text{grid}}}$ and then approximate the derivative operator in (20) by finite differences, for example,

$$\frac{d^2}{dr^2}\mathbf{f}(r_i) = \sum_j C_{ij}\mathbf{f}(r_j) \tag{50}$$

This converts the differential equation into a set of linear equations of the form

$$\Lambda\mathbf{f} = \boldsymbol{\beta} \tag{51}$$

with Λ a band matrix made up of the $\mathsf{D}(r_i)$ and C_{ij}, $\mathbf{f}$ the vector made up

of a particular column of $\tilde{\mathbf{f}}$, say column n_0, at each of the N_{grid} distances, and $\boldsymbol{\beta}$ consists of values of column n_0 of $\tilde{\mathbf{f}}$ for distances less than r_1 and greater than $r_{N_{\text{grid}}}$. For r less than r_1 we set $\tilde{\mathbf{f}}$ to zero. For r greater than $r_{N_{\text{grid}}}$, the boundary conditions are more complex, because they require knowledge of the scattering matrix, which is not available at this step of the calculation. Thus we are restricted to finite difference approximations that require knowledge of the solution only at one distance beyond $r_{N_{\text{grid}}}$. Then an arbitrary normalization can be introduced into (24) and we can take

$$\tilde{f}_{nn_0}(r_{N_{\text{grid}}+1}) = \delta_{nn_0} \tag{52}$$

The bandwidth of Λ depends on the number of points in the finite difference approximation, and in our calculations we have found it efficient to use a nine-point approximation everywhere except for the large r end of the grid where the number of points is reduced to seven and then eventually down to three in order that only one point beyond the end of the grid is required for the boundary conditions. Of course this necessitates using a smaller step size for the last grid points in order that the error of the solution not be dominated by the error in the final three-point approximation.[144] With this scheme the half-bandwidth of Λ is equal to $4N$, where N is the number of channels included in the close-coupling equations.

There are several advantages of this method which offset to a certain degree the extra storage resources required by it. Perhaps the most important feature is the high quality of radial functions produced. In contrast to initial value methods that directly determine the radial functions, the FDBVM has no difficulties with instability due to classically forbidden regions. Thus if rather than just requiring the scattering matrix, one is interested in using the radial functions for other purposes, such as in integrals, then the FDBVM may be the method of choice. An important factor in making this worthwhile is the fact that there is no restriction on the grid points, that is, they can be unevenly spaced. We have used this feature to advantage by including Gaussian quadrature points in the finite difference grid and then used the resulting radial functions in numerical integrals using efficient Gaussian quadrature rules.[144] Another feature of the method is that it is very easy to solve inhomogeneous versions of (20), because the inhomogeneity will simply appear in $\boldsymbol{\beta}$. In addition, because the work to solve (51) is primarily made up of forming the LU decomposition of Λ, it is possible to solve for several inhomogeneities for little extra cost once the homogeneous problem has been solved.[145] The final advantage is that (51) can be solved by reasonably efficient black box programs, so that good execution rates can be obtained even for relatively small N, in con-

trast to the step-by-step methods discussed above. Another aspect of this is that most of the time will be spent solving one set of linear equations, so the user need not be concerned with optimizing large amounts of code, but simply calling the library routine.

We next turn to methods that are not based on (20) but rather on equivalent integral equations. The number of methods here is very large,[146–148] but the basic ideas are similar. Rather than numerically obtaining the radial functions by integrating a differential equation, the radial functions (or related quantities) are expanded in terms of known basis functions, and then the unknown coefficients are determined from the solution of a single set of linear equations, or the scattering matrix and related quantities are determined directly from matrix elements over the basis functions using linear algebra. Thus the work in forming the scattering matrix can be broken down into two parts, the calculation of the matrix elements and the solution of the linear equations. If there is an average of M basis functions per each of the N channels, then for large enough $M \cdot N$, the work for the matrix element calculation will scale as $(M \cdot N)^2$ while the work for the linear equations step will scale as $(M \cdot N)^3$. Thus rather then performing many matrix operations that scale as N^3, there is one large operation scaling as $(M \cdot N)^3$, so the size of M^3 compared to the number of integration steps will be important in determining the relative efficiency of these methods compared to the methods based on differential equations. However, these basis function methods have several advantages, which can counterbalance inefficiencies and make them the methods of choice. A particular example is in the area of rearrangement collisions. The quantum mechanical equations of motion based on the differential equation approach lead in this case to integrodifferential equations,[149] which are very difficult to solve. Although it is possible by a suitable choice of coordinates to turn the rearrangement equations of motion into differential equations,[150] new complexities arise.[151, 152] In contrast, the application of basis function methods leads only to exchange integrals which involve all known functions, so the methodology is virtually unchanged.[144] Another potential advantage is that the linear equations can be solved iteratively, so the work per initial state will scale as $m(M \cdot N)^2$, where m is the number of iterations required.[153–155] This is an advantage when one is interested in transitions out of only a few initial states, so only a few columns of the scattering matrix are needed; however, all of the methods that we have discussed so far use arbitrary boundary conditions and hence require the generation of N linearly independent solutions in order to determine any column of the scattering matrix. Another potential advantage is based on the way scattering calculations are usually performed. In order to assess the convergence of the calculations, it is necessary to perform a sequence of calculations that differ in the number

of basis functions used, and thus many problems that are very similar are solved. What we wish to do is use some of the information from smaller, preliminary calculations to make the larger calculations less expensive. One way to do this is to form linear combinations of the M basis functions based upon smaller calculations and use a smaller number of these contracted functions in the linear equation step.[156] Even if the number of basis functions per channel can be reduced only slightly, the cubic operation count will magnify this reduction and make it a worthwhile step. The parallels here with improving the basis functions in electronic structure calculations (such as the use of atomic natural orbitals) are striking.

These basis function methods are still quite new for applications to heavy particle collisions, and are undergoing rapid development. One important area that needs further work is the efficient evaluation of the matrix elements. While the scaling arguments presented above show that this will be a negligible step for large enough calculations, experience to date is that the coefficient multiplying the $(M \cdot N)^2$ factor is large enough so that it can dominate the calculation. However, the best implementation here is probably yet to appear.

All of this discussion relies on the assumption that the resources for the calculations exist, and this means primarily memory. If a calculation based on the differential equation approach, which only requires the storage of matrices of order N^2, taxes the memory of a machine, clearly the basis function approach is not practical. For this reason, almost all converged calculations of three-dimensional heavy-particle collisions using basis functions have been performed on the CRAY-2.

We now turn to the final method to be considered. Here we move from the time-independent Schrödinger equation to the time-dependent form. This introduces many new complexities into the calculations, but at the same time several advantages appear. The basic idea is that the time-dependent Schrödinger equation is first order in time, so the initial conditions determine the solution for all other times.[157] This in turn requires that the calculations are for a particular initial quantum state, in contrast to most of the methods discussed above. The initial condition consists of a mixed basis function–grid representation of the wave function, and the initial translational energy range is determined via the uncertainty principle from the spatial localization of the wave function. The solution is propagated forward in time until after the collision is over and the results are analyzed to determine the probabilities for the various final states. These calculations are both computationally intensive and consume considerable storage resources, so most large-scale calculations have been carried out on the CRAY-2. There are three main advantages of the method. First, as mentioned above, the method only considers a single initial state. Thus the scaling of the operations of the method will be less

than the third power, unlike the previous methods discussed, and so for large enough calculations this method will be more efficient. Second, since the initial conditions consist of a spread of initial energies, it is possible to obtain results for many different energies from a single time propagation run. Finally, since the time-dependent equation is solved and one is explicitly dealing with wave packets, the physical interpretation of the results can bring more insight than for the time-independent methods.

5.4. The Influence of Supercomputers on Dynamics Calculations

Relatively few classical dynamics studies utilizing vectorized codes running on Cray computers have been reported in the literature, mainly because the calculations can be carried out on slower machines. Indeed, the time per trajectory is small enough that even personal computers have been used. What contributes to the overall time is the number of trajectories required, and the advantage of a vectorized code is that results can be obtained in much less real time (perhaps days rather than months). This offers the opportunity for much better feedback to others whose work depends on the outcome of the dynamics, just as discussed in Section 4 for the electronic structure case.

In the area of quantum dynamics, the advent of supercomputers has opened entire new horizons. The available processing power is beginning to make the study of nonreactive collisions in systems of more than three atoms feasible; this will allow the investigation of phenomena like vibrational-to-vibrational energy transfer, a process that cannot occur for less than four atoms. The large memory of the CRAY-2 has been the main impetus behind a renaissance in quantum reactive scattering, for the application of the basis function methods that have proved so fruitful would not be practical on smaller machines.

6. Conclusions and Future Directions

We have reviewed the performance of essentially all the Cray computers at the time of writing on typical tasks in quantum chemistry and dynamics. Our discussions show that is is possible to achieve a large fraction of the possible machine performance in all phases of these calculations, and we have also considered the impact these developments have had on research in quantum chemistry and dynamics. In view of the fact that one consequence of an increase in available computing power is to whet researchers' appetites for even more computing power, we shall discuss briefly here some future directions.

The most important development in the next few years is likely to be

a much greater use of multitasking on Cray computers. As we have discussed here and elsewhere,[28] multitasking will probably be a necessity to ensure that all CPUs are kept busy in multiple CPU systems, and it should also be used when a single job needs access to a large fraction of system resources. The arrival of microtasked library subroutines should encourage more multitasking, as will the incorporation of autotasking into the CFT77 compiler, but some burden will fall on the programmer if coarse-grained parallelism is to be exploited. There has been little work reported investigating the use of multitasking on normal production machines: our own experiences suggest that some work remains to be done at the operating system level in implementing multitasking. Given that there is the potential on the Y-MP to achieve close to 2.5 GFLOPS using all eight CPUs, as discussed in Section 3.4, the rewards from multitasking can be considerable, and we can expect to see much work in this area in the near future.

Another important development will be the arrival of new supercomputer models. As this article was going to press, it was announced that CRI proposed to divide into two independent companies, one continuing to be called Cray Research, Inc. and the other to be called Cray Computer Corporation (denoted CCC hereafter). CRI will continue to produce the X-MP, Y-MP, and CRAY-2 series machines, and to develop a new supercomputer—the C-90. CCC will concentrate on the development of a new machine, to be called the CRAY-3, and eventually its successor, the CRAY-4. We will discuss briefly the proposed design of both the CRI and CCC machines. CRI's C-90 will have 16 CPUs, each with peak performance of 1 GFLOPS giving 16 GFLOPS total. The main memory will be 512 MW in size and an SSD of up to 2 GW will be available.[158] This appears to a natural extension of the X-MP/Y-MP product line. CCC's CRAY-3, featuring 16 CPUs (clock period 2 ns) using gallium arsenide technology (and 512 MW of silicon memory), has been described in some detail.[159] The theoretical maximum performance would again be 1 GFLOPS per CPU, or 16 GFLOPS total. This machine is an obvious development from the CRAY-2, and much of the experience with the latter should carry over to the CRAY-3. There is already discussion of the CRAY-4[159] from CCC, which is planned to out-perform the original CRAY-1 by a factor of 1,000—three orders of magnitude in some twenty years. Since these machines will have memories of 512 MW and larger, they will stimulate not only the wider use of some of the large-memory-based algorithms we have described in Sections 4 and 5, but also the development of new schemes that can exploit even larger memories. This will undoubtedly lead to novel and exotic approaches to many problems, but the more commonplace methods described earlier will also see substantial performance benefits from the newer machines. It should be noted,

however, that the size of electronic structure problems that can be tackled using conventional methods is commonly limited by external storage capacity even on the current range of Cray computers, and this limitation will only become more acute as faster machines appear. While methods designed to circumvent this limitation will undoubtedly be developed, it is very likely that providing adequate external storage will be a major problem for supercomputer vendors in the next decade.

Having reviewed developments in hardware and software for computational chemistry on Cray computers, a final conclusion can be drawn. The power of Cray supercomputers, with their particular suitability for computational chemistry, has stimulated many important and fundamental developments in the field, and as the next generations of Cray computers appear we can be confident that this trend will continue. There is therefore every reason for optimism about what the rather natural pairing of computational chemistry and Cray supercomputers will bring forth.

Acknowledgments

We would like to acknowledge helpful discussions with many colleagues: J. Almlöf, J. Avila, D. H. Bailey, J. Barton, L. A. Barnes, R. J. Duchovic, T. U. Helgaker, A. Komornicki, S. R. Langhoff, T. J. Lee, H. Partridge, J. E. Rice, and P. Siegbahn. E. Wimmer, and E. A. Kroll of CRI made it possible for us to obtain information about the CRAY-3 and CRAY-4 and R. A. Eades and D. Mason of CRI made numerous helpful comments on the manuscript and provided information about the C-90. N. Davenport provided information about the proposed formation of CCC and on the CRAY-3 and CRAY-4. We would also like to acknowledge the assistance provided by ZeroOne systems, the Advanced Computational Facility, and the Numerical Aerodynamic Simulation Facility at NASA-Ames. We are especially grateful to the NASA Facility for early access to the CRAY Y-MP. PRT was supported by NASA grant No. NCC 2-371.

References

1. *CRAY X-MP and CRAY Y-MP Multitasking Programmer's Manual* (publication No. SR-0222), Cray Research Inc., Mendota Heights, Minnesota (1988).
2. *CRAY-2 Multitasking Programmer's Manual* (publication No. SN-2026), Cray Research Inc., Mendota Heights, Minnesota (1988).
3. *Advances in Chemical Physics*, Vols. 67 and 69, John Wiley, New York (1987).
4. R. B. Bernstein, ed., *Atom–Molecule Collision Theory*, Plenum Press, New York (1979).

5. M. Baer, ed., *Theory of Chemical Reaction Dynamics*, Vols. 1–5, CRC Press, Boca Raton, Florida (1985).
6. R. D. Levine and R. B. Bernstein, *Molecular Reaction Dynamics and Chemical Reactivity*, Oxford University Press, Oxford (1987).
7. *COS Reference Manual* (publication No. SR-0011), Cray Research Inc., Mendota Heights, Minnesota (1987).
8. *UNICOS User Commands Reference Manual* (publication No. SR-2011), Cray Research Inc., Mendota Heights, Minnesota (1988).
9. C. W. Bauschlicher, A. Komornicki, H. Partridge, and P. R. Taylor, unpublished work.
10. M. Metcalf and J. Reid, *FORTRAN 8X Explained*, Oxford University Press, Oxford (1987).
11. *FORTRAN (CFT) Reference Manual* (publication No. SR-0009), Cray Research Inc., Mendota Heights, Minnesota (1987).
12. *CRAY-2 FORTRAN (CFT2) Reference Manual* (publication No. SR-2007), Cray Research Inc., Mendota Heights (1987).
13. *CFT77 Reference Manual* (publication No. SR-0018), Cray Research Inc., Mendota Heights, Minnesota (1987).
14. J. J. Dongarra and S. C. Eisenstat, Squeezing the most out of an algorithm in Cray FORTRAN, *ACM Trans. Math. Software* **10**, 219–230 (1984).
15. *FX/FORTRAN Programmer's Handbook*, Alliant Computer Systems Corp., Littleton, Massachusetts (1987).
16. *FX/SERIES Scientific Library*, Alliant Computer Systems Corp., Littleton, Massachusetts (1987).
17. *Programmer's Library Reference Manual* (publication No. SR-0113), Cray Research Inc., Mendota Heights, Minnesota (1987).
18. V. R. Saunders and M. F. Guest, Applications of the CRAY-1 for quantum chemical calculations, *Comp. Phys. Comm.* **26**, 389–295 (1982).
19. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Software* **5**, 308–323 (1979).
20. J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *Preliminary Proposal for a Set of Level 3 BLAS*, Argonne National Laboratory (1987).
21. R. W. Hockney and C. R. Jesshope, *Parallel Computers—Architecture, Algorithms and Programming*, Adam Hilger, Bristol, England (1980).
22. *Library Reference Manual* (publication No. SR-0014), Cray Research Inc., Mendota Heights, Minnesota (1986).
23. D. A. Calahan, P. L. Berry, G. C. Carpenter, K. B. Elliott, U. M. Fayyad, and C. M. Hsiao, MICHPAK: A scientific library for the CRAY-2, University of Michigan report No. SARL 8 (1985).
24. V. Strassen, Gaussian elimination is not optimal, *Numer. Math.* **13**, 354–356 (1969).
25. V. Pan, New fast algorithm for matrix operations, *SIAM J. Comput.* **9**, 321–342 (1980).
26. D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, *SIAM J. Comput.* **11**, 472–492 (1982).
27. D. H. Bailey, Extra high speed matrix multiplication on the CRAY-2, *SIAM J. Sci. Stat. Comput.* **9**, 603–607 (1988).
28. P. R. Taylor and C. W. Bauschlicher, Strategies for obtaining the maximum performance from current supercomputers, *Theoret. Chim. Acta* **71**, 105–115 (1987).
29. S. R. Bourne, *The UNIX System*, Addison-Wesley, Reading, Massachusetts (1983).
30. C. W. Bauschlicher, Considerations in vectorizing the CI procedure, in *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed.), pp. 13–18, Reidel, Dordrecht (1984).
31. MOLECULE is a vectorized Gaussian integral program written by J. Almlöf.
32. J. Almlöf and P. R. Taylor, unpublished work.
33. SWEDEN is a vectorized SCF-MCSCF, direct CI, conventional CI-CPF-MCPF

program, written by P. E. M. Siegbahn, C. W. Bauschlicher, B. Roos, P. R. Taylor, A. Heiberg, J. Almlöf, S. R. Langhoff, and D. P. Chong.
34. D. J. Fox, Y. Osamura, M. R. Hoffmann, J. F. Gaw, G. Fitzgerald, Y. Yamaguchi, and H. F. Schaefer, Analytic energy second derivatives for general correlated wave functions, including a solution of the first-order coupled-perturbed configuration-interaction equations, *Chem. Phys. Lett.* **102**, 17–22 (1983).
35. J. Almlöf and P. R. Taylor, Molecular properties from perturbation theory: A unified treatment of energy derivatives, *Int. J. Quantum Chem.* **27**, 743–768 (1985).
36. T. U. Helgaker and P. Jørgensen, Analytical calculation of geometrical derivatives in molecular electronic structure theory, *Adv. Quantum Chem.* **19**, 183–245 (1988).
37. *Optimization Guide* (publication No. SN-0220), Cray Research Inc., Mendota Heights (1983).
38. V. R. Saunders, Molecular integrals for Gaussian functions, in *Methods in Computational Molecular Physics* (G. H. F. Diercksen and S. Wilson, ed.), pp. 1–36, Reidel, Dordrecht (1983).
39. D. Hegarty and G. van der Velde, Integral evaluation algorithms and their implementation, *Int. J. Quantum Chem.* **23**, 1135–1153 (1983).
40. D. Hegarty, Evaluation and processing of integrals, in *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed.), pp. 39–66, Reidel, Dordrecht (1984).
41. R. C. Raffenetti, General contraction of Gaussian atomic orbitals: Core, valence, polarization, and diffuse basis sets; molecular integral evaluation, *J. Chem. Phys.* **58**, 4452–4458 (1973).
42. T. H. Dunning and P. J. Hay, Gaussian basis sets for molecular calculations, in *Modern Theoretical Chemistry*, Vol. 3, *Methods of Electronic Structure Theory* (H. F. Schaefer, ed.), pp. 1–27, Plenum, New York (1977).
43. H. Taketa, S. Huzinaga, and K. O-ohata, Gaussian-expansion methods for molecular integrals, *J. Phys. Soc. Japan* **21**, 2313–2324 (1966).
44. S. F. Boys, Electronic wave functions I. A general method of calculation for the stationary states of any molecular system, *Proc. R. Soc. London, Ser. A* **200**, 542–554 (1950).
45. L. E. McMurchie and E. R. Davidson, One- and two-electron integrals over Cartesian Gaussian functions, *J. Comput. Phys.* **26**, 218–231 (1978).
46. S. Obara and A. Saika, Efficient recursive computation of molecular integrals over Cartesian Gaussian functions, *J. Chem. Phys.* **84**, 3963–3974 (1986).
47. J. Almlöf and P. R. Taylor, Computational aspects of direct SCF and MCSCF methods, in *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed.), pp. 107–125, Reidel, Dordrecht (1984).
48. J. Almlöf, University of Stockholm Institute of Physics report No. 74-29 (1974).
49. R. M. Pitzer, Contribution of atomic orbital integrals to symmetry orbital integrals, *J. Chem. Phys.* **58**, 3111–3112 (1973).
50. E. R. Davidson, Use of double cosets in constructing integrals over symmetry orbitals, *J. Chem. Phys.* **62**, 400–403 (1975).
51. J. Almlöf and P. R. Taylor, General contraction of Gaussian basis sets. I. Atomic natural orbitals for first- and second-row atoms, *J. Chem. Phys.* **86**, 4070–4077 (1987).
52. M. Yoshimine, Construction of the Hamiltonian matrix in large configuration interaction calculations, *J. Comput. Phys.* **11**, 449–454 (1973).
53. C. C. J. Roothaan and P. S. Bagus, Atomic self-consistent field calculations by the expansion method, *Meth. Comp. Phys.* **2**, 47–94 (1963).
54. C. W. Bauschlicher and H. Partridge, Vectorizing the sparse matrix vector product on the CRAY X-MP, CRAY-2, and CYBER 205, *J. Comput. Chem.* **8**, 636–644 (1987).
55. F. W. Bobrowicz and W. A. Goddard, The self-consistent field equations for generalized

valence bond and open-shell Hartree–Fock wave functions, in *Modern Theoretical Chemistry*, Vol. 3, *Methods of Electronic Structure Theory* (H. F. Schaefer, ed.), pp. 79–127, Plenum Press, New York (1977).
56. P. S. Bagus, Energy expressions for open shell configurations of linear molecules for use in SCF calculations, IBM report No. RJ 1077 (1972).
57. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines—EISPACK Guide*, Springer-Verlag, Berlin (1976).
58. S. Wilson, Integral transformations, in *Methods of Computational Chemistry*, Vol. 1 (S. Wilson, ed.), Plenum Press, New York (1987).
59. H.-J. Werner, Matrix-formulated direct MCSCF and multireference CI methods, *Adv. Chem. Phys.* **69**, 1–62 (1987).
60. W. Meyer, R. Ahlrichs, and C. E. Dykstra, The method of self-consistent electron pairs. A matrix oriented CI study, in *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed.), pp. 19–38, Reidel, Dordrecht (1984).
61. I. Shavitt, The method of configuration interaction, in *Modern Theoretical Chemistry*, Vol. 3, *Methods of Electronic Structure Theory* (H. F. Schaefer, ed.), pp. 189–275, Plenum Press, New York (1977).
62. P. D. Dacre, On the use of symmetry in SCF calculations, *Chem. Phys. Lett.* **7**, 47–48 (1970).
63. M. Elder, Use of molecular symmetry in SCF calculations, *Int. J. Quantum Chem.* **7**, 75–83 (1973).
64. M. Dupuis and H. F. King, Molecular symmetry and closed-shell SCF calculations, *Int. J. Quantum Chem.* **11**, 613–625 (1977).
65. M. Yoshimine, in IBM report No. RA-18 (ed. W. Lester, 1971).
66. C. F. Bender, Integral transformations. A bottleneck in molecular quantum mechanical calculations, *J. Comput. Phys.* **9**, 547–554 (1972).
67. C. W. Bauschlicher, S. R. Langhoff, P. R. Taylor, N. C. Handy, and P. J. Knowles, Benchmark full CI calculations on HF and NH_2, *J. Chem. Phys.* **85**, 1469–1474 (1986).
68. C. W. Bauschlicher, S. R. Langhoff, and P. R. Taylor, *Adv. Chem. Phys.*, in press.
69. E. R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices, *J. Comput. Phys.* **17**, 87–94 (1975).
70. B. Liu, The simultaneous expansion method for the iterative solution of several of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, in *Report on the NRCC Workshop on Numerical Algorithms in Chemistry: Algebraic Methods* (C. Moler and I. Shavitt, eds.), pp. 49–53, NRCC. Berkeley (1978).
71. P. Siegbahn, A new direct CI method for large CI expansions in a small orbital space, *Chem. Phys. Lett.* **109**, 417–423 (1984).
72. P. J. Knowles and N. C. Handy, A new determinant-based full configuration-interaction method, *Chem. Phys. Lett.* **111**, 315–321 (1984).
73. C. W. Bauschlicher and P. R. Taylor, Benchmark full CI calculations on H_2O, F, and F^-, *J. Chem. Phys.* **85**, 2779–2783 (1986).
74. C. W. Bauschlicher and P. R. Taylor, Benchmark full CI calculations for several states of the same symmetry, *J. Chem. Phys.* **86**, 2844–2848 (1987).
75. P.-Å. Malmqvist, A. P. Rendell, and B. O. Roos, to be published.
76. M. A. Robb and U. Niazi, The unitary group approach in configuration interaction methods, *Comput. Phys. Rep.* **1**, 127–236 (1984).
77. B. O. Roos and P. E. M. Siegbahn, The direct configuration interaction method from

molecular integrals, in *Modern Theoretical Chemistry*, Vol. 3, *Methods of Electronic Structure Theory* (H. F. Schaefer, ed.), pp. 277–318, Plenum Press, New York (1977).
78. P. E. M. Siegbahn, Generalizations of the direct CI method based on the graphical unitary group approach I. Single replacements from a complete CI root function of any spin, first-order wave functions, *J. Chem. Phys.* **70**, 5391–5397 (1979).
79. P. E. M. Siegbahn, Generalizations of the direct CI method based on the graphical unitary group approach II. Single and double replacements from any set of reference configurations, *J. Chem. Phys.* **72**, 1647–1656 (1980).
80. I. Shavitt, Matrix element evaluation in the unitary group approach to the electron correlation problem, *Int. J. Quantum Chem. Symp.* **12**, 5–32 (1979).
81. B. Liu and M. Yoshimine, The ALCHEMY configuration interaction method. I. The symbolic matrix method for determining elements of matrix operators, *J. Chem. Phys.* **74**, 612–616 (1981).
82. P. J. Knowles and H.-J. Werner, An efficient method for the evaluation of coupling coefficients in configuration interaction calculations, *Chem. Phys. Lett.* **145**, 514–522 (1988).
83. P. E. M. Siegbahn, unpublished work.
84. P. R. Taylor, A rapidly convergent CI expansion based on several reference configurations, using optimized correlating orbitals, *J. Chem. Phys.* **74**, 1256–1270 (1981).
85. R. Ahlrichs, A new MR-CI(SD) technique, in *Proceedings of the 5th Seminar on Computational Methods in Quantum Chemistry, Groningen*, 1981 (P. Th. van Duijnen and W. C. Nieuwpoort, eds.), pp. 254–272, Max-Planck-Institut, Garching bei München (1981).
86. V. R. Saunders and J. H. van Lenthe, The direct CI method: A detailed analysis, *Mol. Phys.* **48**, 923–954 (1983).
87. B. O. Roos, The CASSCF method and its application in electronic structure calculations, *Adv. Chem. Phys.* **69**, 399–445 (1987).
88. P. E. M. Siegbahn, J. Almlöf, A. Heiberg, and B. O. Roos, The complete active space SCF (CASSCF) method in a Newton–Raphson formulation with application to the HNO molecule, *J. Chem. Phys.* **74**, 2384–2396 (1981).
89. B. O. Roos, P. R. Taylor, and P. E. M. Siegbahn, A complete active space SCF method (CASSCF) using a density matrix formulated super-CI approach, *Chem. Phys.* **48**, 157–173 (1980).
90. B. O. Roos, The complete active space SCF method in a Fock-matrix-based super-CI formulation, *Int. J. Quantum Chem. Symp.* **14**, 175–189 (1980).
91. B. O. Roos, unpublished work.
92. J. Almlöf and H. P. Lüthi, Theoretical methods and results for electronic structure calculations on very large systems, in *Supercomputer Research in Chemistry and Chemical Engineering*. ACS Symposium Series 353 (K. F. Jensen and D. G. Truhlar, eds.), pp. 35–48, American Chemical Society, Washington, D. C. (1987).
93. T. U. Helgaker, J. Almlöf, H. J. Aa. Jensen, and P. Jørgensen, Molecular Hessians for large-scale MCSCF wave functions, *J. Chem. Phys.* **84**, 6266–6279 (1986).
94. T. U. Helgaker, H. J. Aa. Jensen, P. Jørgensen, and P. R. Taylor, ABACUS, an MCSCF analytic energy derivatives program.
95. J. Almlöf, K. Faegri, and K. Korsell, Principles for a direct SCF approach to LCAO-MO ab initio calculations, *J. Comput. Chem.* **3**, 385–399 (1982).
96. M. Feyereisen and J. Almlöf, unpublished work.
97. M. Häser and R. Ahlrichs, Improvements on the direct SCF method, *J. Comput. Chem.* **10**, 104–111 (1989).
98. P. R. Taylor, Integral processing in beyond-Hartree–Fock calculations, *Int. J. Quantum Chem.* **31**, 521–534 (1987).
99. S. Saebø and J. Almlöf, Avoiding the integral storage bottleneck in LCAO calculations, *Chem. Phys. Lett.* **154**, 83–89 (1989).

100. M. Head-Gordon, J. A. Pople, and M. Frisch, MPZ energy evaluation by direct methods, *Chem. Phys. Lett.* **153**, 503–506 (1989).
101. B. Liu and A. D. McLean, *Ab initio* potential curve for Be_2 from the interacting correlated fragments method, *J. Chem. Phys.* **72**, 3418–3419 (1980).
102. M. R. A. Blomberg and P. E. M. Siegbahn, The ground-state potential curve for F_2, *Chem. Phys. Lett.* **81**, 4–13 (1981).
103. K. Jankowski, R. Becherer, P. Scharf, H. Schiffer, and R. Ahlrichs, The impact of higher polarization basis functions on molecular *ab initio* results. I. The ground state of F_2, *J. Chem. Phys.* **82**, 1413–1419 (1985).
104. S. R. Langhoff, C. W. Bauschlicher, and P. R. Taylor, Accurate *ab initio* calculations for the ground states of N_2, O_2, and F_2, *Chem. Phys. Lett.* **135**, 543–548 (1987).
105. C. W. Bauschlicher, S. R. Langhoff, and P. R. Taylor, On the $^1A_1$–$^3B_1$ separation in CH_2 and SiH_2, *J. Chem. Phys.* **87**, 387–391 (1987).
106. D. L. Cochrane and D. G. Truhlar, Strategies and performance norms for efficient utilization of vector pipeline computers as illustrated by the classical mechanical simulation of rotationally inelastic collisions, *Parallel Comput.* **6**, 63–88 (1988).
107. D. W. Schwenke, Calculations of rate constants for the three-body recombination of H_2 in the presence of H_2, *J. Chem. Phys.* **89**, 2076–2091 (1988).
108. R. Bulirsch and J. Stoer, Numerical treatment of ordinary differential equations by extrapolation methods, *Numer. Math.* **8**, 1–13 (1966).
109. J. T. Muckerman, Applications of classical trajectory techniques to reactive scattering, *Theor. Chem.: Adv. Perspec.* **6A**, 1–77 (1981).
110. H. Goldstein, *Classical Mechanics*, Addison-Wesley, Reading, Massachusetts (1980).
111. L. L. Poulsen, G. D. Billing, and J. I. Steinfeld, Temperature dependence of HF vibrational relaxation, *J. Chem. Phys.* **68**, 5121–5127 (1978).
112. W. A. Lester, Coupled-channel studies of rotational and vibrational energy transfer by collision, *Adv. Quantum Chem.* **9**, 199–214 (1975).
113. M. S. Child, *Molecular Collision Theory*, Academic Press, London (1984).
114. J. M. Blatt and L. C. Biedenharn, The angular distribution of scattering and reaction cross sections, *Rev. Mod. Phys.* **24**, 258–272 (1952).
115. D. G. Truhlar, C. A. Mead, and M. A. Brandt, Time-reversal invariance representations for scattering wavefunctions, symmetry of the scattering matrix, and differential cross sections, *Adv. Chem. Phys.* **33**, 295–344 (1975).
116. W. A. Lester, Calculation of cross sections for rotational excitation of diatomic molecules by heavy particle impact: solution of the close-coupling equations, *Meth. Comp. Phys.* **10**, 211–241 (1973).
117. *Meth. Comp. Phys.* **10** (1971).
118. *Comp. Phys. Commun.* **6** (6), (1973).
119. *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory* (L. Thomas, ed.), National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley, California (1979).
120. M. H. Alexander, Hybrid quantum scattering algorithms for long-range potentials, *J. Chem. Phys.* **81**, 4510–4516 (1984).
121. M. H. Alexander and D. E. Manolopoulos, A stable linear reference potential algorithm for solution of the quantum close-coupled equations in molecular scattering theory, *J. Chem. Phys.* **86**, 2044–2050 (1987).
122. D. W. Schwenke and D. G. Truhlar, A new potential energy surface for vibration-vibration coupling in HF–HF collisions. Formulation and quantal scattering calculations, *J. Chem. Phys.* **88**, 4800–4813 (1988).
123. D. W. Schwenke, A new method for the direct calculation of resonance parameters with application to the quasibound states of the $H_2 X^1\Sigma_g^+$ system, *Theoret. Chim. Acta* **74**, 381–402 (1988).

124. J. C. Light and R. B. Walker, An *R*-matrix approach to the solution of coupled equations for atom–molecule reactive scattering, *J. Chem. Phys.* **65**, 4272–4282 (1976).
125. E. B. Stechel, R. B. Walker, and J. C. Light, *R*-matrix solution of coupled equations for inelastic scattering, *J. Chem. Phys.* **69**, 3518–3531 (1978).
126. D. G. Truhlar, N. M. Harvey, K. Onda, and M. A. Brandt, Applications of Close-Coupling Algorithms to Electron–Atom, Electron–Molecule, and Atom–Molecule Scattering, in *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory* (L. Thomas, ed.), pp. 220–289, National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley, California (1979).
127. G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore (1983).
128. S. T. Elbert, Extracting more than a few eigenvectors from a dense real symmetric matrix: Optimal algorithms versus the architectural constraints of the FPS-X64, *Theoret. Chim. Acta* **71**, 169–186 (1987).
129. J. J. Dongarra, L. Kaufman, and S. Hammarling, *Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers*, Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 46 (1985).
130. D. W. Schwenke and D. G. Truhlar, Converged calculations of rotational excitation and V-V energy transfer in the collision of two molecules, in *Supercomputer Simulations in Chemistry* (M. Dupuis, ed.), pp. 165–197, Springer-Verlag, Berlin (1986).
131. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia (1979).
132. W. A. Lester, DeVogelaere's method, in *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory* (L. Thomas, ed.), pp. 105–115, National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley, California (1979).
133. M. E. Riley and A. Kuppermann, Vibrational energy transfer in collisions between diatomic molecules, *Chem. Phys. Lett.* **1**, 537–538 (1968).
134. D. W. Schwenke, K. Haug, D. G. Truhlar, R. H. Schweitzer, J. Z. H. Zhang, Y. Sun, and D. J. Kouri, Storage management strategies in large-scale quantum dynamics calculations, *Theoret. Chim. Acta* **72**, 237–251 (1987).
135. J. M. Launay, Body-fixed formulation of rotational excitation: Exact and centrifugal decoupling results for CO-He, *J. Phys. B* **9**, 1823–1838 (1976).
136. D. W. Schwenke and D. G. Truhlar, An optimized quadrature scheme for matrix elements over the eigenfunctions of general anharmonic potentials, *Comp. Phys. Commun.* **34**, 57–66 (1984).
137. N. A. Mullaney and D. G. Truhlar, The use of rotationally and orbitally adiabatic basis functions to calculate rotational excitation cross sections for atom–molecule collisions, *Chem. Phys.* **39**, 91–104 (1979).
138. J. M. Launay, Molecular collision processes I. Body-fixed theory of collisions between two systems with arbitrary angular momenta, *J. Phys. B* **10**, 3665–3672 (1977).
139. G. Gioumousis and C. G. Curtiss, Molecular collisions. II. Diatomic molecules, *J. Math. Phys.* **2**, 96–104 (1961).
140. D. W. Schwenke, D. G. Truhlar, and M. E. Coltrin, Comparison of close coupling and quasiclassical trajectory calculations for rotational energy transfer in the collision of two HF molecules on a realistic potential energy surface, *J. Chem. Phys.* **87**, 983–992 (1987).
141. J. V. Lill, G. A. Parker, and J. C. Light, Discrete variable representations and sudden models in quantum scattering theory, *Chem. Phys. Lett.* **89**, 483–489 (1982).
142. D. G. Truhlar and A. Kuppermann, Exact tunneling calculations, *J. Am. Chem. Soc.* **93**, 1840–1851 (1971).
143. J. Z. H. Zhang, D. J. Kouri, K. Haug, D. W. Schwenke, Y. Shima, and D. G. Truhlar,

$\mathscr{L}^2$ amplitude density method for multichannel inelastic and rearrangement collisions, *J. Chem. Phys.* **88**, 2492–2512 (1988).
144. D. W. Schwenke, K. Haug, M. Zhao, D. G. Truhlar, Y. Sun, J. Z. H. Zhang, and D. J. Kouri, Quantum mechanical algebraic variational methods for inelastic and reactive molecular collisions, *J. Phys. Chem.* **92**, 3202–3216 (1988).
145. D. W. Schwenke, M. Mladenovic, M. Zhao, D. G. Truhlar, Y. Sun, and D. J. Kouri, Computational strategies and improvements in the linear algebraic variational approach to rearrangement scattering, in *Supercomputer Algorithms for Reactivity, Dynamics, and Kinetics of Small Molecules* (A. Laganà, ed.), pp. 131–168, Kluwer, Dordrecht (1988).
146. D. G. Truhlar, J. Abdallah, and R. L. Smith, Algebraic variational methods in scattering theory, *Adv. Chem. Phys.* **25**, 211–293 (1974).
147. G. Staszewska and D. G. Truhlar, Convergence of $\mathscr{L}^2$ methods for scattering problems, *J. Chem. Phys.* **86**, 2793–2804 (1987).
148. J. Z. H. Zhang, S.-I. Chu, and W. H. Miller, Quantum scattering via the *S*-matrix version of the Kohn variational principle, *J. Chem. Phys.* **88**, 6233–6239 (1988).
149. W. H. Miller, Coupled equations and the minimum principle for collisions of an atom and a diatomic molecule, including rearrangements, *J. Chem. Phys.* **50**, 407–418 (1969).
150. D. W. Schwenke, D. G. Truhlar, and D. J. Kouri, Propagation method for the solution of the arrangement-channel coupling equations for reactive scattering in three dimensions, *J. Chem. Phys.* **86**, 2772–2786 (1987).
151. R. T. Pack and G. A. Parker, Quantum reactive scattering in three dimensions using hyperspherical (APH) coordinates. Theory, *J. Chem. Phys.* **87**, 3888–3921 (1987).
152. G. Schatz, Quantum reactive scattering using hyperspherical coordinates: Results for $H + H_2$ and $Cl + HCl$, *Chem. Phys. Lett.* **150**, 92–98 (1988).
153. L. Thomas, Solution of the coupled equations of inelastic atom–molecule scattering for a single initial state. II. Use of nondiagonal matrix Green's functions, *J. Chem. Phys.* **76**, 4925–4931 (1982).
154. B. I. Schneider and L. A. Collins, Direct iteration-variation method for scattering problems, *Phys. Rev. A* **33**, 2970–2981 (1986).
155. C. Duneczky, R. E. Wyatt, D. Chatfield, K. Haug, D. W. Schwenke, D. G. Truhlar, Y. Sun, and D. J. Kouri, Iterative methods for solving the non-sparse equations of quantum mechanical reactive scattering, *Comp. Phys. Commun.* **53**, 357–379 (1989).
156. J. Abdallah and D. G. Truhlar, Use of contracted basis functions in algebraic variational scattering calculations, *J. Chem. Phys.* **61**, 30–36 (1974).
157. Y. Sun, R. C. Mowrey, and D. J. Kouri, Spherical wave close-coupling wave packet formalism for gas phase nonreactive atom–diatom collisions, *J. Chem. Phys.* **87**, 339–349 (1987).
158. R. A. Eades, private communication.
159. S. Cray, presentation at CRI annual general meeting, 1988.

3

Chemical Calculation on Japanese Supercomputers

K. MOROKUMA, U. NAGASHIMA, S. YAMAMOTO, N. KOGA, S. OBARA, AND S. YABUSHITA

1. Introduction

Though some users previously had used computers with optional integrated array processor (IAP), serious supercomputing in Japan started only in late 1983, when the first Hitac S-810 was installed at the University of Tokyo Computer Center. A Facom VP-100 was introduced soon afterward to Nagoya and Kyoto Universities. The Computer Center of the Institute for Molecular Science (IMS), which provides several thousand hours of CPU time to the molecular science community, started the service of a Hitac S-810/10 supercomputer in January, 1986, together with a powerful scalar mainframe Hitac M-680H. The IMS supercomputer was upgraded to a brand new Hitac S-820/80 in February, 1988. The present equipment at the IMS Computer Center is shown in Figure 1.

As of January, 1989, universities and academic institutions in Japan have the following supercomputers in operation: four Hitac S-820's, four Facom VP-400 or 200's, three NEC SX-2's, and one ETA-10.

Since our experience has been mostly limited to Japanese supercomputers, and little is known about them outside Japan, we will first

K. MOROKUMA, U. NAGASHIMA, S. YAMAMOTO, AND N. KOGA • Institute for Molecular Science, Myodaiji, Okazaki 444, Japan. S. OBARA • Department of Chemistry, Kyoto University, Kyoto, Japan. S. YABUSHITA • Department of Chemistry, Hiroshima University, Hiroshima, Japan.

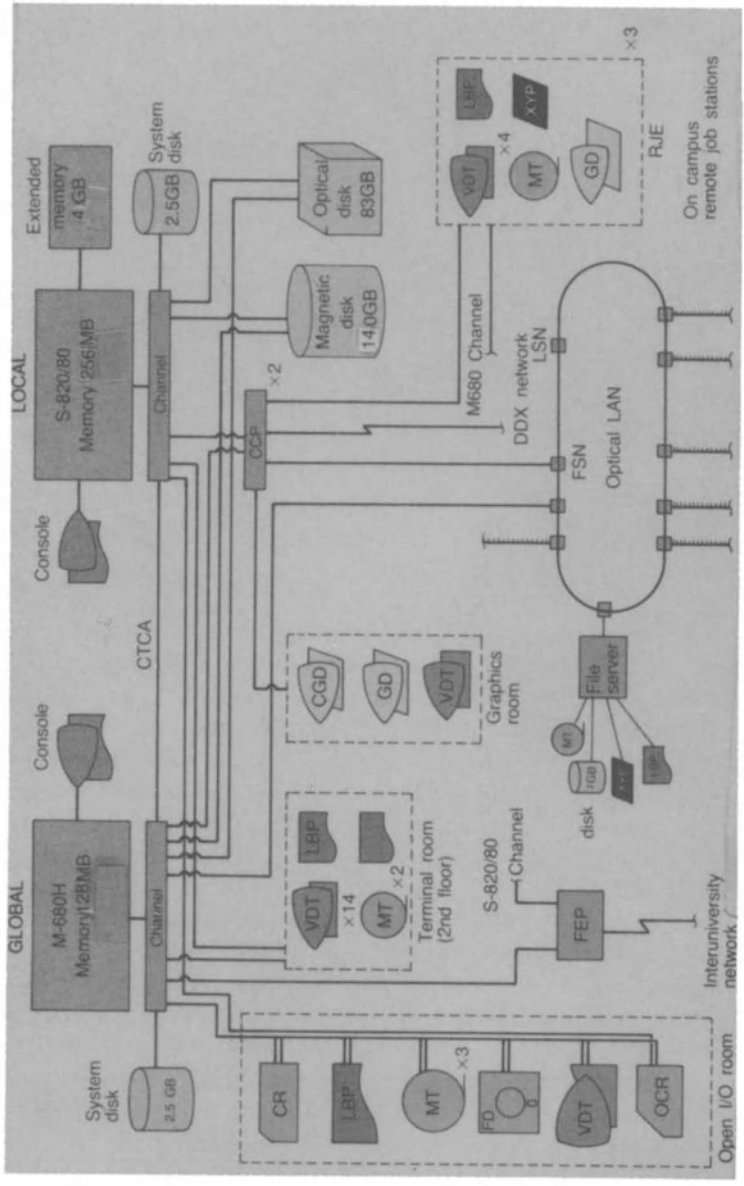


Figure 1. Present computers at the Computer Center of the Institute for Molecular Science (memory sizes in parentheses, as of 1989).

describe briefly characteristics of Japanese supercomputers. Then we will discuss our supercomputer experiences in molecular electronic structure calculations.

2. Japanese Supercomputers

2.1. Hardware

The models of supercomputers available from Japanese manufacturers are Hitac S-820/80 and 60 and S-810/20, 10, and 5 from Hitachi, Facom VP-400E, 200E, 100E, 50E, and 30E from Fujitsu, and NEC SX-2, 1, 1E, and J from NEC. Their hardware characteristics are shown in Table 1, together with those of some Cray computers for comparison. Fujitsu has recently announced new VP-2000 series with a maximum peak speed of 4 GFLOPS (GFLOPS = 10^9 floating-point operations per second).

Their common features can be summarized as follows:

1. Very high speed central processing unit (CPU), with several parallel pipelines.
2. In addition to large main memory, Hitac and NEC have a large capacity of transistor extended memory, used in place of disk storage during execution. Facom has a large vector storage.
3. Single-processor machine. No multiple-processor machine has been announced as of January 1989.

We have found at IMS that the extended memory is essential for supercomputing. By eliminating essentially all the I/O time, most of the jobs we run on a supercomputer become CPU-bound and allow a very efficient use of CPU time.

The basic performance of these machines has been compared in Figure 2 with 14 Livermore loop kernels. On the average the ratio of the speed of these machines in this test is roughly S-820/80:SX-2:VP-400:X-MP/1 = 1:1/2:1/3:1/7.

2.2. Software

The operating systems (OS), FORTRAN compilers, and vectorization tools available on Japanese supercomputers are summarized in Table 2. All of the operating systems are "IBM-VM-like," with the exception of the NEC machines, which employ a different job control. All of the compilers are highly intelligent with automatic vectorizing capabilities and can

Table 1. Hardware Characteristics of Some Supercomputers[a]

Machine	Model	Machine cycle (ns)	Peak speed (MFLOPS)	Pipeline speed (MFLOPS)	Number of pipes	Multiplication speed/CPU (MFLOPS)	Number of CPUs	Maximum main memory (MB)	Maximum extended memory (GB)
Hitac	S-820/80	4.0	3000	250	12	1000	1	512	12
	60	4.0	1500	250	6	500	1	256	6
	S-810/20	14.0	852	71.4	12	286	1	256	3
	10	14.0	426	71.4	6	143	1	128	3
	5	14.0	213	71.4	3	71	1	128	3
Facom	VP-400E	7.0	1700	142.8	12	857	1	256	(768)[b]
	200E	7.0	850	142.8	6	430	1	256	(768)
	100E	7.0	425	142.8	3	210	1	256	(256)
	50E	7.0	280	142.8	3	135	1	256	(256)
	30E	7.5	220	142.8	3	105	1	128	(128)
NEC	SX-2	6.0	1300	166.6	8	666.6	1	256	8
	1	7.0	570	142.8	4	285.6	1	256	8
	1E	7.0	285	142.8	2	142.8	1	128	8
	J	9.6	210	142.8	1	104.5	1	128	8
Cray	X-MP/4	8.5	940	117.6	2	117.6	4	512 × 4	—
	2/4	4.1	1950	243.9	2	243.9	4	32 × 4	4

[a] Sources: Y. Karaki, *Computrol* **20**, 2 (1987) and data from manufacturer's catalogs. As of January 1989.
[b] Vector memory in MB in parentheses.

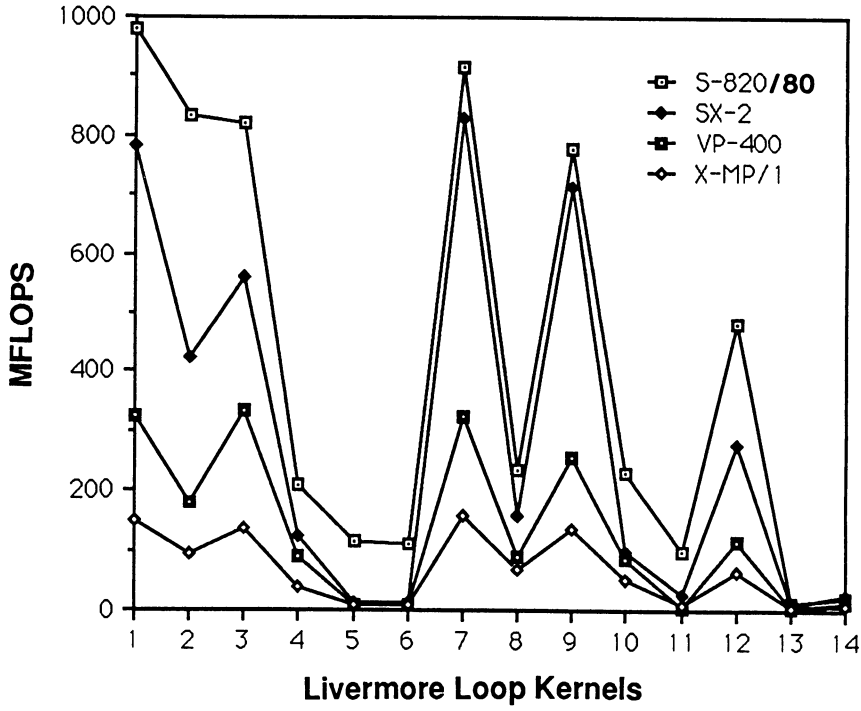


Figure 2. Performance of supercomputers on the 14 Livermore Loop kernels. Source: C. Eoyang, *Vector Register*, Vol. 1, No. 3 (1988), Recruit Institute For Supercomputer Research.

vectorize first-order sequential operations, indirect addressing (list vector), the IF statement, mathematical functions, logical and integer operations, as well as real number operations.

Japanese supercomputers are equipped with convenient tools which facilitate optimum vectorization and program tuning. They analyze the structure of the program and give various data such as the loop or statement that has not been vectorized and the reason for it, the ratio of

Table 2. Software for Japanese Computers

	Hitac S	Facom VP	NEC SX
OS	VOS3/HAP/ES	OSIVF4/MSP + VCPF VSP(backend)	SXCP
Compiler	FORT77/HAP	FORTRAN77/VP	FORTRAN77/SX
Vectorization tools	FORT/ASSIST FORT VF	FORTUNE Conversational vectorizer	Analyzer/SX Vectizer/SX Optimizer

vectorized versus unvectorized codes, and the time spent in each segment of a program. It also gives suggestions as to how to modify the source code to improve vectorization, or it even gives an improved new source code. We find that these tools are very useful and indispensable in our practice of optimizing vector codes.

3. Applications

3.1. Linear Algebra Subroutines

Subroutines of linear algebra play a very important role in several stages of quantum chemical calculation. Though supercomputers are very good at linear algebra, the refinement of algorithms and coding techniques are quite important to achieve a high efficiency. Here, in this section we introduce our attempts for matrix multiplication and diagonalization.

3.1.1. General Comments

Two types of mathematical subroutine packages are available on supercomputers. One is provided by computer manufacturers and the other is developed by computer users. Since the former is developed with a full knowledge of both assembler language and individual computer architecture, the performance of these subroutines is generally excellent on each corresponding supercomputer. They are SSL II/VP, MATRIX/HAP, ASL/SX and MXM for Facom VP, Hitac S-810 and S-820, NEC SX and Cray X-MP, respectively. Their subroutines cover a wide range of mathematical problems from linear algebra to numerical integrations and special functions. For numerical accuracy, they are quite good because higher precision operations are applied to many numerically sensitive parts. Unfortunately, their source codes are not usually open to the users, so that debugging is hard if an error occurred in one of their subroutines.

Subroutine packages developed by computer users are also useful, because they are usually written in FORTRAN with source codes made available and are easily debugged and checked for accuracy. The performance of this kind of subroutines depends strongly on the intelligence of the FORTRAN compiler. As pointed out in the previous section, intelligent FORTRAN compilers and useful program-tuning tools are available on each computer, and the performance of user-developed subroutines has also been excellent for the calculation of linear algebra.

The NUMPAC developed by Ninomiya and Hatano[1] is one of this type of packages. The NUMPAC has been widely installed and used in most computer centers of universities and academic institutions in Japan.

This program package has been tuned up to individual supercomputers, so that there are three NUMPAC versions: NUMPAC/VP, NUMPAC/S820, and NUMPAC/SX.

3.1.2. Matrix Multiplication

To compare the performance of matrix multiplication of various algorithms and codes, Kanda and Nagashima[2] measured the CPU time of two-dimensional matrix multiplication on Facom VP-400, NEC SX-2, Hitac S-820/80, and Cray X-MP/216.

Table 3. Codes for Matrix Multiplication

Inner product

```
   DO 10 J=1,N
     DO 10 I=1,N
       A(I,J)=0.0
     DO 10 K=1,N
10     A(I,J)=A(I,J)
         +B(I,K)*C(K,J)
```

Middle product

```
     DO 10 J=1,N
       DO 10 I=1,N
10       A(I,J)=0.0
       DO 12 J=1,N
         DO 12 K=1,N
           DO 12 I=1,N
12           A(I,J)=A(I,J)
               +B(I,K)*C(K,J)
```

Outer product

```
     DO 10 J=1,N
       DO 10 I=1,N
10       A(I,J)=0.0
       DO 12 K=1,N
         DO 12 J=1,N
           DO 12 I=1,N
12           A(I,J)=A(I,J)
               +B(I,K)*C(K,J)
```

Type of matrix multiplication

```
Type a : A(I,J)=A(I,J)+B(I,K)*C(K,J)      A=B*C
     b : A(I,J)=A(I,J)+B(K,I)*C(K,J)      A=B(trans)*C
     c : A(I,J)=A(I,J)+B(I,K)*C(J,K)      A=B*C(trans)
     d : A(I,J)=A(I,J)+B(K,I)*C(J,K)      A=B(trans)*C(trans)
```

Unrolling (of multiplicity 2)

```
   DO 10 J=1,N
     DO 10 I=1,N
10     A(I,J)=0.0
   DO 12 J=1,N                    ------------------->
     DO 12 K=1,N
       DO 12 I=1,N
12       A(I,J)=A(I,J)
           +B(I,K)*C(K,J)
```

```
     MM=N/2*2
     DO 60 J=1,N
       DO 10 I=1,N
10       A(I,J)=0.0
     DO 30 K=1,MM,2
       DO 20 I=1,N
20       A(I,J)=A(I,J)+B(I,K)*C(K,J)
           +B(I,K+1)*C(K+1,J)
30   CONTINUE
     DO 50 K=MM+1,N
       DO 40 I=1,N
40       A(I,J)=A(I,J)+B(I,K)*C(K,J)
50   CONTINUE
60   CONTINUE
```

There are three product codes for matrix multiplication: inner product, middle product, and outer product, as shown in Table 3, depending on the loop position of the product index K. In addition, there are four types of products one can make. The efficiency of 12 codes, all requiring the same number of floating point operations, are examined on Hitac S-820/80 and the results are summarized in Table 4. Two sizes M of the matrix dimension, 1001 and 1024, are used to study the effect of bank conflict. Table 4 shows that the conflict is more serious on vector computing. In the case of $M = 1024$, reflecting the bank conflict, the performance falls down on both scalar and vector computing. If one wants to choose only one type of product, the outer product is the most stable and fastest for all types of matrix multiplication, with a scalar versus vector ratio of over 100. The middle product is slightly less stable, but is not far behind the outer product.

Comparisons of performance of matrix multiplication codes on various supercomputers are shown in Table 5. FORTRAN codes are examined only for type b inner, type a middle, and type c outer products, the fastest in each product in Table 4. The effect of unrolling is also examined with multiplicity of 2 and 8 by using the type a middle product. Loop unrolling technique is known to work well on supercomputers that have multiple pipelines. Details of unrolling is also shown in Table 3. In Table 5, column 4 represents the FORTRAN subroutine packages provided by the manufacturers for a scalar machine but compiled with a vector code. The

Table 4. CPU Time of Various Codes on Hitac S-820/80 (s)

Loop $N=999$	Type	Scalar			Vector			Ratio (S/V)	
		M[a] = 1024	1001	Ratio	1024	1001	Ratio	1024	1001
Inner product	a[b]	592.85	308.41	1.92	34.05	3.50	9.72	17.41	88.12
	b[b]	115.07	116.87	0.98	3.34	3.03	1.10	34.45	38.57
	c[b]	1086.67	732.52	1.48	34.05	3.38	10.07	31.91	216.72
	d[b]	603.41	306.10	1.97	34.91	3.64	9.59	17.29	84.09
Middle product	a	148.08	144.05	1.02	1.22	1.23	0.99	121.48	117.11
	b	606.89	327.55	1.85	20.19	1.35	14.95	30.07	242.63
	c	147.74	144.83	1.02	1.51	1.32	1.14	97.52	109.71
	d	635.13	346.54	1.83	20.04	1.38	14.52	31.69	251.11
Outer product	a	148.69	144.93	1.02	1.37	1.25	1.10	108.53	115.94
	b	637.60	351.22	1.81	5.41	1.33	4.07	118.07	264.08
	c	146.69	143.34	1.02	1.22	1.25	0.98	120.23	114.67
	d	660.57	332.44	1.98	5.38	1.33	4.05	122.90	249.95

[a] M, the size of the dimension.
[b] See Table 3.

Table 5. CPU Time for Matrix Multiplication (for $N = 999$)[a]

Computer and FORTRAN version	Dimension size M	SSL2/VP ASL/SX MATRIX/HAP MXM	SSL2/M MATH/SX MSL2	Inner Product type b	Middle Product type a	Outer Product type c	Unrolling multiplicity[b] 2	Unrolling multiplicity[b] 8
VP-400								
FORT77/VP	1024	1.816	97.967	97.368	1.807	1.830	1.993	1.944
V10130	1001	1.913	3.647	3.598	1.842	1.860	1.977	1.946
SX-2								
FORT77/SX	1024	1.905	81.485	3.702	3.470	3.470	3.028	1.971
REV. 038	1001	1.909	4.048	3.776	3.566	3.570	3.061	1.973
S-820/80								
FORT77/HAP	1024	1.303	16.682	3.338	1.233	1.224	1.206	1.222
22-00	1001	1.304	1.689	3.032	1.234	1.243	1.205	1.214
Cray X-MP/216								
CFT77	1024			13.150	14.753	14.348	10.477	10.137
3.0	1001	9.121[c]		13.345	14.786	14.447	10.905	10.216

[a] M, the size of the dimension. N, the actual size of the square matrix = 999.
[b] Middle product of type a.
[c] The size of the dimension M is 999 because of the subroutine restriction.

performance of maker-provided vector subroutines is reasonable. The relative speed of computers at this level is 1(S-820/80):1/1.5 (SX-2 and VP-400):1/7 (X-MP/2). Since the intelligence of Japanese FORTRAN compilers is quite high, the difference between the maker-provided subroutines and the FORTRAN programs is not large, except for SX-2 where the maker-provided subroutines outperformed the FORTRAN codes by a factor of nearly 2.

3.1.3. Matrix Diagonalization

For diagonalization of a packed matrix, supercomputers show a very impressive power. Beppu and Ninomiya[3] developed new codes for diagonarization of a real symmetric dense matrix. According to their measurement on several matrices, their code based on the Housholder + QR method is the most efficient on a Japanese supercomputer. The speed-up ratio from scalar to vector calculation is about 30 for their new code. The results on Hitac S-820/80 are summarized in Table 6. Their program has been implemented into some SCF programs in the IMS computer Center Program Library to improve the supercomputer performance.

For a large sparse matrix that often appears in CI calculation, Kosugi's modification[4] of the Davidson–Liu method[5, 6] allows one to obtain simultaneously several eigensolutions, either the lowest or higher ones, without knowledge of the exact lower solutions, and has been coded[4] to give a very high performance. For example, the speed-up ratio (S/V) of CPU time is about 17 in the CI calculation of 61113 Slater determinants for the SiF_4 molecule with a DZP basis set.

3.2. Integral Evaluation

Computation of molecular integrals still remains a very important problem in *ab initio* calculation with a large number of basis functions. In an early work by Kosugi,[7] a better vectorization of the Gauss–Rys code was achieved based on the original Saunders–Guest algorithm.[8] His code works about three times faster than on a scalar computer.

A more impressive work has recently been carried out by Obara and Saika.[9] They proposed and derived recurrence expressions for various types of molecular integrals over Cartesian Gaussian functions by using a recurrence formula for three-center overlap integrals, leading to an efficient computing method. Their code ran 5 times faster than HONDO even on a scalar computer, and additionally 4 or 5 times on a vector computer.

To analyze the efficiency of the method in more detail, the CPU time was measured separately for the integral evaluation part and for the other

Table 6. CPU Time (s) on Hitac S-820/80 for Diagonalization to Obtain All Eigenvalues and Eigenvectors ($N = M = 256$)

	Scalar				Vector				Ratio (S/V)			
	Frank	Block	Random	Hückel	Frank	Block	Random	Hückel	Frank	Block	Random	Hückel
EISQL	5.096	5.557	6.721	5.918	0.412	0.441	0.508	0.457	13.34	13.57	14.03	13.70
HDES1M	—[a]	—	—	—	0.371	0.445	0.396	0.402	—	—	—	—
NSHOUD	5.268	6.152	7.271	6.344	0.144	0.177	0.224	0.187	36.58	34.76	32.46	33.93
HOQRVW	4.716	5.776	6.735	5.788	0.144	0.184	0.228	0.189	32.75	31.39	29.54	30.62

[a] HSDES1M works only for vector computation.

```
EISQL:EISPAC,       HOUSHOLDER+QL
DSEIG1:SSL II/VP,    HOUSHOLDER+QL
HDES1M:MATRIX/HAP,  HOUSHOLDER+BISECTION+INVERSE ITERATION
NSHOUD:NUMPAC,      HOUSHOLDER+QR, new code
HOQRVW:NUMPAC,      HOUSHOLDER+QR, new code

Definition of matrices;
Frank matrix  F(i,j)=N+1-MAX(i,j)
Block matrix  B(i,j)=4*delta(i,j)-sum(k=1,r-1){sum(l=0,r-2){/B}}
              where r=sqrt(n);
                    /B=delta(i,k+1+rl)*delta(j,i+r+1)
                       +delta(i,k+rl)*delta(j,i+r+1)
                       +delta(j,k+1+rl)*delta(i.j+r-1)
                       +delta(j,k+rl)*delta(i,j+r+1)
                    delta(a,b)=0  a NE b
                              =1  a=b

                    sum(k=a,b){S(k)}: summation S(k) from k=a to b
Random matrix     -32768<R(i,j)<+32768
Hückel matrix  H(i,j)=-7.2delta(i,j)-3.0/(i-j)**2*(1-delta(i,j))
```

part, including indexing, labeling, buffering, and I/O. In Table 7, results on Facom VP-400 are shown. The integral evaluation part of KOTO, Obara's integral code,[10] is well vectorized and shows high performance on the vector computer. For Gaussian80 and HONDO, automatic vectorization caused worsening of their performance because of short loop lengths. The difference in CPU time required for labeling, which is not vectorized at all, is small between the three codes. Since the integral evaluation step of KOTO is so efficient on a vector computer, one is actually spending more time in labeling than in evaluation. For instance, in the case of C_6H_{12}(MIDI-1), the CPU time for labeling is about 10 times longer than for evaluation. Since this tendency would be enhanced with larger and higher-quality basis sets, the implementation of a direct SCF code in conjunction with the Obara routine has to be seriously considered.

Table 7. CPU Time of Integral Evaluation (E) and Labeling (L)[a] on Facom VP-400

Molecule and basis set	Program	Scalar(S) / Vector(V)	E (s)	L (s)	Ratio (E/L)
C_2H_4O MIDI-4	GAUSSIAN80[b]	S	8.938	2.020	4.42
		V	15.570	1.948	7.99
		Ratio (S/V)	0.57	1.03	
	HONDO[b]	S	34.190	0.951	35.95
		V	48.116	0.950	50.64
		Ratio (S/V)	0.71	1.00	
	KOTO[c]	S	7.094	0.731	9.70
		V	0.372	0.788	0.47
		Ratio (S/V)	19.06	0.92	
C_6H_{12} MINI-1	KOTO	S	61.066	1.642	37.19
		V	2.179	1.763	1.23
		Ratio (S/V)	28.2	0.93	
C_6H_{12} MIDI-1	KOTO	S	65.667	18.648	3.62
		V	3.534	20.537	0.17
		Ratio (S/V)	18.58	0.90	

[a] All the CPU time except for the integral evaluation step is included in L.
[b] Only automatic vectorization by the FORTRAN compiler was used for vector execution.
[c] Obara's integral program.[10]

Obara's integral evaluation code has been implemented in some library programs of IMS computer Center Library, including Gaussian 82 and 86.

3.3. Self-Consistent Field Calculations

In the iterative procedure of SCF calculations, the generation of the Fock matrix is the most time-consuming step. Efforts have been made to reduce the CPU time required in the Fock matrix generation. When the two-electron integrals are stored as the PK supermatrix,[11] the number of operations is reduced several times, compared with the original form of two-electron integrals (rs/tu). Kosugi discussed vectorization of Fock matrix generation in detail using the sorted PK integrals.[7] He has shown that vectorization using the sorted PK integrals accelerates the SCF cycles drastically.

Here, let us consider only the RHF calculations, for simplicity. Only the P supermatrix is necessary for RHF Fock matrix generation.

```
P(rs,tu)=(rs/tu)-{(ru/st)+(rt/su)}/4
   where rs=r*(r-1)/2+s and tu=t*(t-1)/2+u
   and r>=s, t>=u, and rs>=tu
```

Using the P integrals, the Fock matrix is calculated as follows:

```
  Read P(rs,tu) with labels, RS(rs) and TU(tu).
  DO 1 I=1,N
  rs=RS(I)
  tu=TU(I)
  F(rs)=F(rs)+D(tu)*P(rs,tu)
1 F(tu)=F(tu)+D(rs)*P(rs,tu),
```

where N is the number of nonzero integrals stored and D and F are the linearized Fock and density matrix modified for the use of PK integrals.

When the P integrals are stored randomly, the vectorization of this DO loop is impossible. However, if P integrals are stored so that tu runs with fixed rs, the Fock matrix generation can be vectorized. Then, the structure of the DO loop looks like this:

```
  DO 1 rs=1,M*(M+1)/2
  Read Prs(tu) with labels TU(tu).
  DO 2 tu=1,Ntu
  F(rs)     =F(rs)     +D(TU(tu))*Prs(tu)
2 F(TU(tu))=F(TU(tu))+D(rs)     *Prs(tu)
1 Continue
```

where M is the number of atomic orbitals and Ntu is the number of nonzero integrals Prs(tu) for a specified rs.

In loop 2, the specific element of tu appears once per one value of rs. Therefore, the vectorization of DO loop can be forced by a compiler option. Though this example is for RHF Fock matrix generation, the extension is straightforward to UHF, ROHF, and even GVB.

In Table 8 is shown the timing data by Kosugi[7] and by us. The acceleration rates from vectorization for entries 1–3 are larger than 6, being quite good. On the other hand, one should note that in Gaussian82 its original assembler routines were used for scalar computation. The assembler (scalar) to FORTRAN (vector) acceleration rate is good.

The acceleration rates of UHF calculation obtained in the case of C_2H_5 is larger than that of RHF calculation. The number of operations in the DO Loop for UHF Fock matrix generation is twice as large as that for RHF, resulting in a better acceleration.

The critical step thus becomes sorting of the PK integrals. When the PK integrals are already stored in a disk, some sorting algorithms, for instance, Yoshimine's two pot algorithm, are available. Yoshimine's algorithm is carrying out read- and write-operations only two times and thus is not so time-consuming. A more convenient method, however, has been proposed by Kosugi, in which a canonically ordered PK file is generated directly by the AO integral program.[7] The algorithm is shown below:

```
 DO 1 r=1,M    M is the number of AOs.
 DO 2 s=1,r
 DO 2 t=1,s
 DO 2 u=1,t
   evaluation of (rs/tu), (ru/st) and (rt/su)
2 P(s,tu)=(rs/tu)-{(ru/st)+(rt/su)}/4
   Write P integrals in an external storage so that tu runs
   with fixed s.
1 Continue
```

The outer four DO loops run over atomic orbitals to simplify discussion, though in the actual program they run over "shells." After the loop 2 is completed, the sorted P integrals are stored in an external storage. The direct formation of the sorted PK integrals has been implemented in Kosugi's GSCF3[12] and the IMS modification of Gaussian82 and has turned out to be efficient. For instance, the original Gaussian82 requires 30.1 s of CPU time for evaluation of integrals for C_2H_5, whereas the modified routines involving the Kosugi algorithm takes 32.8 s, a negligible increase.

Table 8. Comparison of CPU Time (s/cycle) of SCF Calculations on Hitac S-810

	Scalar	Vector	*S/V*	Reference
C_2H_5OH[a]	1.2	0.2	6.0	7[h]
SiF_4[b]	3.0	0.4	7.1	7[h]
SNi_5[c]	83.3	7.7	10.9	7[h]
CH_3OH[d]	0.4[g]	0.2	2.8	13[i]
C_2H_5[e]	2.3[g]	0.4	5.2	13[i]
$Rh_2Cp_2(H)(CH_3)(CH_2)_2$[f]	18.6[g]	4.2	4.4	13[i]

[a] C_s, 4-31G*, 60 CGTOs.
[b] C_{2v}, Si[5321/521/1], F[621/51/1], 79 CGTOs.
[c] C_{2v}, S[5212111/411111/111], Ni[53321/5311/311], 218 CGTOs.
[d] C_s, 6-311G*, 48 CGTOs.
[e] C_s, 6-311G*, 51 CGTOs.
[f] Rh[21/21/31], C, H, STO-2G, 123 CGTOs.
[g] Assembler routines were used.
[h] Program system GSCF3 coded by Kosugi.
[i] GAUSSIAN82 and its modifications.

3.4. Four-Index Transformation

A new program named JASON2[14] has been developed at IMS to perform *ab initio* CASSCF calculations with large basis sets. The bottleneck in large MCSCF calculations is the time-consuming integral transformation step. This problem has been overcome by Roos *et al.*'s density-averaged super-CI CASSCF method[15] and by the newly developed transformation algorithm.[14] The super-CI method needs only (MMMN)-type transformed integrals. Here M runs over active MOs and N over the entire MOs. The new transformation algorithm makes the best use of this merit. A high speed has been obtained by reducing the total number of operations

Table 9. CPU Timing (s) of CASSCF Calculation for AlO (62 CTGOs, 7 Active MOs, 784 CSFs, C_1 Symmetry)

	JASON2		GAMESS
Step	S-810/10 vector	S-820/80 vector	S-820/80 scalar
SORT	24.1	8.0	—
TR (transformation)	4.3	1.7	64.3
CI	11.4	4.8	7.2
SX (super-CI)	4.3	1.4	24.2[a]
TR + CI + SX/cycle	20.0	7.9	95.7

[a] Newton–Raphson procedure.

based on the sparsity of AO-based integrals and by vectorization. Details of the algorithm are given in Ref. 14.

In Table 9, we present the timing data of the CASSCF calculation of the AlO molecule.[14] The SORT step in Table 9 is sorting of AO-based integrals in order to vectorize the transformation step. This step is required only once prior to CASSCF iterations. The other figures are CPU time consumed in each cycle of the CASSCF calculation. A comparison has been carried out with the program GAMESS, where only the scalar processor was used since automatic vectorization actually slows down the execution speed.

The timing data of a CASSCF calculation of an iron–porphine–oxygen–pyridine complex, FeP(py)O, with 232 CGTOs are shown in Table 10.[16] In an *ab initio* calculation with a large basis set, huge fast I/O facilities are required unless a direct-SCF type method is adopted. In the IMS supercomputer system, we are equipped with magnetic disks of 85 GB, of which 40 GB are devoted to parallel I/O. A sequential file on 16-volume disk set is accessed through 16 separate I/O channels with speed about 40 times as fast as the conventional disk I/O. This parallel I/O disk set, as well as the extended storage discussed in Section 2, has been used extensively in this type of calculation.

3.5. Configuration Interaction

Yabushita[17] has recently adopted the COLUMBUS direct-CI program,[18] originally written for Cray, to IMS's Hitac S-810/S-820. Though he could convert the whole set of programs in a few days, the

Table 10. CPU Timing (s) of CASSCF Calculation for FeP(py)O (232 CGTOs, 10 Active MOs, 5220 CSFs, C_{2v} Symmetry)

	S-810/10	S-820/80		
Step	*V*	*S*[a]	*V*	*S/V*
SORT	649	277	277	1.0
TR	114	191	43	4.4
Step-1	95	139	29	4.8
Step-2	5	30	1	23.7
Step-3	3	7	1	6.0
Step-4	1	5	0	17.6
CI	233	181	107	1.7
SX	66	27	21	1.3
TR + CI + SX/cycle	413	399	172	2.3

[a] The same program, but compiled to use only the scalar processor.

Table 11. CPU Time (s) for CI Calculation with COLUMBUS for the C_4 Molecule

Molecule: C_4 (linear) Scheme: SRCI/TZDP basis functions: 96 CSFs: 122, 243			C_4 (rhombus) MR (SDT') CI/DZP 60 323, 928	
	S-810/10	Cray X-MP	S-810/10	Cray X-MP
4-Ex	7.7	6.6	8.5	8.3
3-Ex	2.4	2.1	8.4	11.0
2-Ex	11.8	6.5	100.6	94.9
1-Ex	1.0	0.9	63.8	62.5
0-Ex	0.2	0.1	20.2	15.2
Total/iter	23.1	16.2	201.5	191.9

Table 12. CPU Time (s) for CI Calculation with COLUMBUS for ICN and C_2H_4

Linear ICN MRCI/ECP + DZP
basis functions: 41

CSF's	$X^1\Sigma^+$ 76,473		$A^3\Pi$ 323,528	
	S-810/10	S-820/80	S-810/10	S-820/80
4-Ex	2.0	1.1	8.6	5.0
3-Ex	2.3	1.0	13.9	6.1
2-Ex	10.4	5.1	109.6	53.5
1-Ex	7.8	3.9	68.7	34.4
0-Ex	2.8	1.4	25.2	12.0
Total/iter	25.3	12.5	226.0	111.0

C_2H_4 MR(SDT')CI/DZP
No. of basis functions: 52
No. of CSF's: 1,046,758

	S-810/10	S-820/80
4-Ex	37.4	22.2
3-Ex	37.2	18.2
2-Ex	484.9	232.5
1-Ex	205.5	105.3
0-Ex	47.6	23.3
Total/iter	812.6	401.5

largest problem was in the linear algebra. In Cray one routinely calls subroutines such as MXM, AXPY, and DOT for highest performance. Hitac, on the other hand, vectorizes the source code in FORTRAN, and therefore, algebra subroutines are not as conveniently prepared, and some rewriting was necessary. Some timing data for Hitac S-810/10 versus Cray X-MP are given in Table 11. The converted COLUMBUS ran slightly more slowly on S-810/10 than on Cray X-MP. Table 12 shows a comparison of S-810/10 versus S-820/80. S-820 in these examples ran about twice as fast as S-810.

4. Concluding Remarks

Starting rather late in supercomputing, quantum chemists in Japan learned a lot from the experiences of American and European colleagues who long had access to CDC and Cray supercomputers. Based on these experiences, we have been able to make demands on some desirable hardware features as well as on the need of an efficient automatic vectorizing compiler. With new features that are realized, our tactics for supercomputing had to be somewhat different. We have shown above that we understand the problems, and can find a way to vectorize essential steps of quantum chemistry calculations. A high efficiency has been attained for several molecular orbital packages. But there are still many codes to be vectorized. Nobody seems to know exactly where Japanese supercomputers are heading from here. In conjunction with some national projects such as "the fifth generation computer project," there are several possible avenues to choose from. As one of the largest supercomputer user groups, quantum chemists in Japan are quite eager to have some input into the future direction of supercomputing.

References

1. I. Ninomiya and Y. Hatano, NUMPAC, mathematical subroutine package. A library program of the IMS Computer Center (1984).
2. K. Kanda and U. Nagashima, unpublished.
3. Y. Beppu and I. Ninomiya, unpublished.
4. N. Kosugi, *J. Comp. Phys.* **55**, 426 (1984); N. Kosugi, EMOR1, a library program of the IMS Computer Center (1986).
5. E. R. Davidson, *J. Comp. Phys.* **14**, 34 (1975); *J. Phys.* **A13**, L179 (1980).
6. B. Liu, The simultaneous expansion method, in *Numerical Algorithms in Chemistry: Algebraic Methods* (C. Moler and I. Shavitt, eds.), Lawrence Berkeley Laboratory, University California, Berkeley, p. 49 (1978).

7. N. Kosugi, *Theoret. Chim. Acta* **72**, 149 (1987).
8. V. R. Saunders and M. F. Guest, Applications of the Cray-1 for quantum chemistry calculations, in *The First International Conference on Vector and Parallel Processors in Computational Science* (P. Burke and L. Delves, eds.), Comput. Phys. Comm. 26, North-Holland, Amsterdam, p. 389 (1982).
9. S. Obara and A. Saika, *J. Chem. Phys.* **84**, 3963 (1986); **86**, 1540 (1988).
10. S. Obara, KOTO, a library program of the IMS Computer Center (1989).
11. R. C. Raffenetti, *Chem. Phys. Lett.* **20**, 335 (1973).
12. N. Kosugi, *Center News* (Computer Center, University of Tokyo) **16**, 74 (1984); **17**, 28 (1985); **17**, 90 (1985).
13. N. Koga, unpublished.
14. S. Yamamoto, U. Nagashima, T. Aoyama, and H. Kashiwagi, *J. Comput. Chem.* **8**, 627 (1988).
15. B. O. Roos, P. R. Taylor, and P. E. M. Siegbahn, *Chem. Phys.* **48**, 157 (1980).
16. S. Yamamoto, J. Teraoka, and H. Kashiwagi, *J. Chem. Phys.* **88**, 303 (1988).
17. S. Yabushita, unpublished.
18. H. Lischka, R. Shepard, F. B. Brown, J. Simons, and I. Shavitt, *Int. J. Quantum Chem. Symp.* **15**, 91 (1984).

4

Quantum Chemical Methods for Massively Parallel Computers

MICHAEL E. COLVIN, ROBERT A. WHITESIDE,
AND HENRY F. SCHAEFER III

1. Introduction to Parallel Computers and Molecular Quantum Mechanics

1.1. Introduction

In 1946 shortly after the development of the first electronic computer John Von Neumann was asked to consult the Office of Naval Research on whether enough applications could be found for high-speed computing devices to justify further funding. Von Neumann concluded:

> It is, furthermore, quite clear that there are many problems where the length or the complexity of the problem is not sufficient to justify electronic speeds. It is important, however, to point out that there are plenty of problems which justify this speed, and, furthermore, the chances are that if these speeds become available, we will come to discover more and more how numerous these problems are.[1]

In the 40 years since Von Neumann made this comment computers have had a revolutionary impact on nearly all aspects of human endeavor. Today computers are essential tools in many fields including business, medicine, publishing, and manufacturing. However, no field has been affec-

MICHAEL E. COLVIN • Lawrence Livermore National Laboratory, Livermore, California 94550. ROBERT A. WHITESIDE • Sandia National Laboratory, Livermore, California 94550. HENRY F. SCHAEFER III • University of Georgia, Athens, Georgia 30602.

ted more dramatically than scientific research. Computers are a ubiquitous presence in the laboratory, where they control instruments and collect and analyze data. Yet the most important function of computers in science is that which Von Neumann originally predicted: as a tool for the accurate theoretical modeling of physical systems.

The potential of such computational models was quickly recognized by scientists, and a number of significant research projects were carried out on the earliest available computers.[2,3] In the subsequent decades scientific computation has had a history of impressive achievements and spawned the creation of many new fields of theoretical research. A large factor in the increasing accuracy and utility of scientific computation has been the phenomenal growth of computer performance. In the past 40 years the instruction speed and memory capacity of computers have increased by nearly six orders of magnitude.

Unfortunately single-processor (or serial) performance will not continue to improve at this rate. Despite a 30-year history of a factor of 10 increase in instructon speeds every 5–7 years, the past decade has seen only a threefold increase in the clock rates of high-end supercomputers.[4] Moreover, fundamental physical constraints such as the speed of light limit single-processor computational speeds to less than about three billion floating point operations per second,[5] which is less than an order of magnitude faster than currently available supercomputers. Such a limitation would seriously impair many fields where the size and accuracy of the systems modeled are severely constrained by available computer speeds.

A possible solution to this problem is to harness together many processors to work on a single computational problem. In principle the net processing speed of such a system is limited only by the number of processors linked together. The concept of so-called parallel processing computers has been around since the earliest days of automatic computing, and a number of experimental parallel computers were proposed and built in the 1960s and 1970s.[6–8] Yet, parallel computers have become commercially available only in the last few years with the development of inexpensive and reliable processing elements. Currently there are more than a dozen commercially announced parallel computers ranging from two processor machines operating at less than a MFLOP (million floating-point operations per second)[9] to 4000 processor machines operating at nearly 100 GFLOPS (billion floating-point operations a second).[10]

The potential capabilities of parallel computers have stimulated tremendous interest in the computer science community (illustrated by the fact that a 1983 bibliography on parallel computers contained a total of 5161 entries).[11] Despite this great enthusiasm among computer scientists, there has been relatively little interest from the physical sciences. This lack of interest is due to the experimental nature of most of the available

parallel computers and the difficulty of programming efficiently such machines. Nevertheless there have been a few significant efforts worthy of note, all the result of collaboration between physical scientists and computer scientists.

One of the earliest such efforts was carried out at Carnegie–Mellon University and involved the development of monte carlo and molecular dynamics algorithms for the 50 processor CM* computer.[12] Two large parallel computing projects are currently underway at IBM and Caltech. The IBM project is aimed at developing quantum chemistry and molecular dynamics algorithms for parallel computers involving relatively small numbers of high-end processors.[13] The Caltech project involves the application of very large arrays of microprocessors to a wide variety of problems in the physical and biological sciences.[14] In the near future the number and magnitude of parallel scientific computation projects will certainly grow, as parallel hardware and expertise become more widespread and the need for faster computers more acutely felt.

1.2. Computers and Molecular Quantum Mechanics

With the development of quantum mechanics in the 1920s it became possible in principle to predict theoretically any chemical property. Such a prediction requires solving the molecular Schrödinger equation, a partial differential equation of the elliptic type which is insoluble for all but trivial cases. Hence, for 30 years theoretical chemists were limited to very small systems (such as helium atom[15] or hydrogen molecule[16]) or to very simplified models of large compounds.[17] The advent of the electronic computer rapidly expanded the horizons of the quantum chemist. In 1956 Boys reported the first quantum chemical calculation carried out entirely on an electronic computer,[3] and only three years later Robert Mulliken predicted: "colossal rewards lie ahead from large scale quantum mechanical calculations of the structure of matter."[18] Since 1959 great strides have been made in this direction. Theoretically predicted molecular properties are used routinely to aid in the interpretation of experimental results, and there have been many instances where discrepancies between theory and experiment have been resolved in favor of theory.[19]

Despite this history of successes, Mulliken's vision of obtaining "the electronic eigenfunction and energy of every major type of molecule" has not been realized. The problem is not that there are inherent limitations in the theoretical methods; these methods could, in principle, be applied to macromolecules or solids. Instead the limitation is due to the inherent complexity and associated computational cost of solving the molecular Schrödinger equation. Even the fastest available supercomputers are not sufficiently fast to allow the study of molecules with more than a few dozen

atoms. Hence, the limitations on single processor performance discussed in the last section will seriously constrain the size of systems amenable to study by quantum chemical methods. If quantum chemical calculations are to be carried out on significantly larger systems, parallel computers will have to be used.

The conversion of standard quantum chemical methods to parallel computers will not be a simple task. Most of these methods require large data sets and involve tightly coupled iterative algorithms not easily broken down into concurrent procedures. Moreover, the quantum chemical procedures have been optimized to make efficient use of single processor computers, further complicating the conversion to parallel machines.

A research group at IBM under the direction of Enrico Clementi has successfully implemented a parallel scheme for doing SCF calculations[20] and is currently developing parallel algorithms for other quantum chemical techniques.[21] However, these algorithms are designed for parallel computers having ten or fewer very large-scale processors. In order to gain the very dramatic performance increases promised by parallel computers, algorithms must be designed to run efficiently on arbitrarily large arrays of processors. The work described in this chapter represents a first step towards this goal. This work involved the design and implementation of parallel algorithms for several major quantum chemical procedures. These algorithms were implemented on a 32 processor Intel Hypercube, a prototype of the much larger parallel computers that will be available in the next few years.

The remainder of this section is divided into three subsections. The first is a description of the methods and computational requirements of molecular quantum mechanics. Following this is a brief overview of the types of parallel computers now available and a description of the Intel Hypercube. The third section is a discussion of general principles for parallelizing scientific programs. Section 2 contains a detailed description of the parallel algorithms, and the third section gives the benchmark results and an analysis of the algorithm efficiencies. The final section discusses quantum chemical methods and computer technologies that seem promising for the future.

1.3. Methods of Molecular Quantum Mechanics

The goal of molecular quantum mechanics is to calculate from first principles the chemical properties of molecular systems. This amounts to solving the molecular Schrödinger equation to determine the wave function of the molecule of interest. From this wave function all observable properties of the molecule can be calculated.

The molecular Schrödinger equation cannot be solved exactly except

for the simplest systems, so that a series of approximations must be made. For most chemical applications, relativistic effects are sufficiently small that they can be ignored.[22] Also, terms coupling nuclear motion to the electronic structure can be omitted from the molecular Hamiltonian (Born–Oppenheimer approximation[23]) so that the problem reduces to determining the electronic wave function for a fixed nuclear framework.

In most quantum chemical calculations the electronic wave function is represented by an antisymmetrized product of molecular orbitals (MOs) known as a Slater determinant:

$$\Psi = \frac{1}{\sqrt{n!}} \det(\phi_1 \phi_2 \cdots \phi_n) \tag{1}$$

These molecular orbitals are approximated as linear combinations of atomic orbitals (AOs):

$$\phi_i = \sum_{\mu} C_{\mu i} \chi_{\mu} \tag{2}$$

The matrix C relating the molecular orbitals to the atomic orbital basis functions is determined by the iterative self-consistent field (SCF)[24] procedure. The atomic orbital basis functions are represented by analytic functions having the form of linear combinations of Gaussian functions:

$$\chi_{\mu} = \sum_{i} d_i x^{l_i} y^{m_i} z^{n_i} e^{-\alpha_i r^2} \tag{3}$$

The contraction coefficients d are fixed before the computation begins. Typically a quantum chemical calculation involves the use of 50–100 of these basis functions.

The SCF procedure is usually carried out in two steps. The first is the numerical calculation of integrals over the basis functions. The most time consuming of these integrals have the form

$$(\mu\nu \mid \lambda\sigma) = \iint \phi_{\mu}(1)\, \phi_{\nu}(1) \frac{1}{r_{12}} \phi_{\lambda}(2)\, \phi_{\sigma}(2)\, d\tau_1\, d\tau_2 \tag{4}$$

These integrals have an inherent eightfold symmetry with respect to the permutation of their indices:

$$(\mu\nu \mid \lambda\sigma) = (\nu\mu \mid \lambda\sigma) = (\lambda\sigma \mid \mu\nu)$$

For a system with n basis functions there are $n_4/8$ of these integrals, hence the computational complexity of this step is $O(n^4)$. The second step is the calculation of the self-consistent field energy and molecular orbitals. This

iterative procedure requires the repeated construction and diagonalization of a Hamiltonian matrix from the precalculated integral list. This Hamiltonian is called the Fock operator and has the form

$$F_{\mu\nu} = H_{\mu\nu} + \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu\,|\,\lambda\sigma) - \tfrac{1}{2}(\mu\lambda\,|\,\sigma\nu)] \tag{5}$$

where **H** is the sum of the kinetic energy and nuclear attraction integrals, and **P** is the density matrix formed from the SCF vector **C**. The complexity of this step is also $O(n^4)$ since the integral list must be processed each iteration. An alternative approach is used when there is insufficient memory to store the two-electron integrals. In this strategy, known as direct SCF, the integral list is recalculated every SCF iteration.

Although many properties calculated from the SCF wave functions are quantitatively accurate,[24] it is often necessary to go beyond this first-order approximation. There are a variety of different methods for extending the accuracy of the SCF wave function. The more widely used methods include: configuration interaction,[25] Moller–Plesset perturbation theory,[26] and coupled-cluster methods.[27] These techniques have the common feature that they all require the AO integral list to be transformed into the MO basis. This transformation has the form

$$(ij\,|\,kl) = \sum_{\mu}\sum_{\nu}\sum_{\lambda}\sum_{\sigma} C_{\mu i} C_{\nu j} C_{\lambda k} C_{\sigma l} (\mu\nu\,|\,\lambda\sigma) \tag{6}$$

where i, j, k, l, are MO indices, $\mu, \nu, \lambda, \sigma$, are AO indices, and **C** is the MO coefficient matrix. As written above, this transformation has complexity $O(n^8)$, but it can be rearranged into a series of partial transformations so that it has complexity $O(n^5)$.[28]

The two most widely used post-SCF techniques are configuration interaction (CI) and Moller–Plesset perturbation theory (MPPT). Both methods have strengths and weaknesses so that they can be used in complementary roles.

Moller–Plesset perturbation theory is standard perturbation theory approach where the zeroth-order Hamiltonian is the Hartree–Fock wave function. The perturbation term is the difference between this approximate Hamiltonian and the full molecular Hamiltonian. When the perturbation expansion is carried out corrections to the SCF energy and wave function to various orders are obtained. The energy expressions have the form of sums of products of transformed integrals and SCF orbital energies. For example the second-order energy correction is

$$E_0^{(2)} = \frac{1}{4} \sum_{abrs} \frac{|\langle ab|\ |rs\rangle|^2}{\varepsilon_a + \varepsilon_b - \varepsilon_r - \varepsilon_s} \tag{7}$$

where ε_a is the SCF energy of orbital a and $\langle ab| \, |rs\rangle$ are the so-called superintegrals formed from the transformed integrals as follows:

$$\langle ab| \, |cd\rangle = (ac \,|\, bd) - (ad \,|\, bc) \tag{8}$$

The energy corrections are routinely calculated to fourth order in quantum chemical calculations.

In the configuration interaction technique the molecular wave function is approximated as a linear combination of Slater determinants. In general the wave function of any molecular system can be written as

$$|\Phi\rangle = c_0 \,|\Psi_0\rangle + \sum_{ra} c_a^r \,|\Psi_a^r\rangle + \sum_{\substack{a<b \\ r<s}} c_{ab}^{rs} \,|\Psi_{ab}^{rs}\rangle + \cdots \tag{9}$$

where $|\Psi_0\rangle$ is the Hartree–Fock wave function and $|\Psi_a^r\rangle$ represents the ground state wave function with an electron excited from the occupied orbital a to virtual orbital r.

The CI energies and wave functions are the eigenvalues and eigenvectors of the CI Hamiltonian matrix:

$$\langle \Phi| \, H \, |\Phi\rangle \tag{10}$$

where H is the molecular Hamiltonian and $|\Phi\rangle$ is the wave function given above. Since the number of terms in the CI wave function grows exponentially in the number of basis functions and electrons, the expansion is usually truncated to some preset excitation level. The most often used CI wave function includes only single and double excitations, which for most systems recovers more than 90% of the correlation energy.[24]

Many interesting chemical properties depend not on the total molecular energies (calculated by the methods just described), but instead depend on the derivative of this energy with respect to nuclear coordinates or some external field. Although these derivatives can be approximated by finite difference techniques from the energy calculations, in the past decade more accurate and efficient methods have been developed. These methods involve explicitly differentiating the energy expressions and then evaluating these exact expressions. These so-called "analytic" derivative methods have been implemented for the first and second derivatives (with respect to nuclear coordinates) for SCF,[29] CI,[30] and MPPT[31] wave functions.

This chapter describes algorithms and implementation results for the integral evaluation, SCF energy calculation, integral transformation, and second-order Moller–Plesset energy calculation. Additionally, an algorithm is described for a parallel configuration interaction program.

1.4. Introduction to Parallel Processors

A 1985 survey by the Parallel Processing Research Council found more than 50 ongoing projects developing new parallel computer architectures.[32] Although these computers all involve the interconnection of many processors, the nature of the individual processors and how they are linked together vary greatly form design to design. Despite the vast number of proposed designs it is possible to classify nearly all parallel computers into a relatively small number of categories.

One broad categorization is made on the basis of whether the individual processors carry out the same or different instruction streams. One design is labeled single instruction multiple data stream (SIMD) and refers to computers where a number of processing elements carry out the same instructions in lock-step on parallel streams of data. SIMD machines are well suited for applications such as numerical solution of differential equations where the identical numerical computation is carried out on each distinct grid point. The classic example of a SIMD machine is the Illiac IV,[7] which contained an 8×8 grid of processors. A recent and more exotic SIMD computer is the "Connection Machine,"[33] which contains 65,536 one-bit processors and was designed for both scientific and artificial intelligence applications. The limitation of SIMD computers is that they can be used efficiently only when the computational problems well match the uniform structure of the machines.

The alternate category is the multiple instruction multiple data stream (MIMD) computers. MIMD describes any linked collection of processing elements that are not constrained to carry out identical instruction streams (as are SIMD machines). To perform useful computation the processors have to be able to communicate, and it is this communication scheme that defines the subcategories of MIMD computers.

The two most common communication strategies are for the processors to share a common pool of memory (shared memory) or to communicate explicitly by sending and receiving message packets (distributed memory). Many of the proposed and currently available MIMD computers use shared memory. In most shared memory computers any processor can access any value in the common memory. This makes programming shared memory computers relatively straightforward, and for this reason most research on automatic parallelizing compilers (i.e., compilers that convert serial programs to parallel) is targeting shared memory computers.

There is, however, a drawback to the shared memory architecture. As the number of processors becomes large the communication with the shared memory can become a bottleneck. It is not yet known how restrictive this bottleneck will prove to be, but most such machines have less than 20 processing elements. A possible way to avoid the limitation is to have

a hierarchy of common memories in which small clusters of processors share a common memory and each cluster has a single channel to a global shared memory.[34] Of course, the introduction of such a hierarchy is at the cost of programming simplicity which was the prime motivation for using the shared memory architecture.

Distributed memory architectures have the advantage that there are in principle no hardware bottlenecks since communication between one pair of processors is independent of communication between another pair. The disadvantage of distributed memory systems is that it is more difficult to program since passing data requires synchronizing both the sending and receiving processors.

An important design parameter of distributed memory computers is the connection topology of the processing elements—that is, the way in which the processors are linked by communication channels. A large variety of communication topologies have been studied. These are usually regular grids in one or more dimensions (where the grid's vertices represent processing elements and the edges represent communication channels). The conclusion of these studies has been that the optimal processor connectivity is dictated by the computational problem at hand.[8] Hence, one option is to build a special purpose processor for each problem to be solved. A more practical alternative is to use a complex connectivity that has a large number of useful subtopologies.

An example of the second alternative is the hypercube architectures. In a hypercube the processor connectivity is that of a n-dimensional Boolean hypercube. A hypercube of dimension n has 2^n processors each connected to n neighbors. Hypercubes of dimension 1, 2, 3, and 4 are shown in Figure 1. This connection topology allows the embedding of regular meshes of lower dimension than the hypercube using a subset of the connections of the hypercube. Such mappings are easily carried out using a technique known as Gray coding.[35]

The goal of the work described in this chapter is to demonstrate the feasibility of very dramatic speedups through the use of parallel computers. Such performance increases require the use of very large numbers of processors, which are currently available only in distributed memory computers. For this reason the algorithms presented were designed with regard to the advantages and limitations of distributed memory machines.

The implementation of the programs was carried out on an Intel IPSC-1 parallel computer (hereafter referred to as the Hypercube) located at Sandia National Laboratory in Livermore, California. The Hypercube is a 32-processor distributed memory computer with hypercube connectivity. In addition to these 32 "node" processors, there is a host processor with a communication channel to each node processor. Each processing node contains an INTEL 80286 CPU and an 80287 floating point processor as well

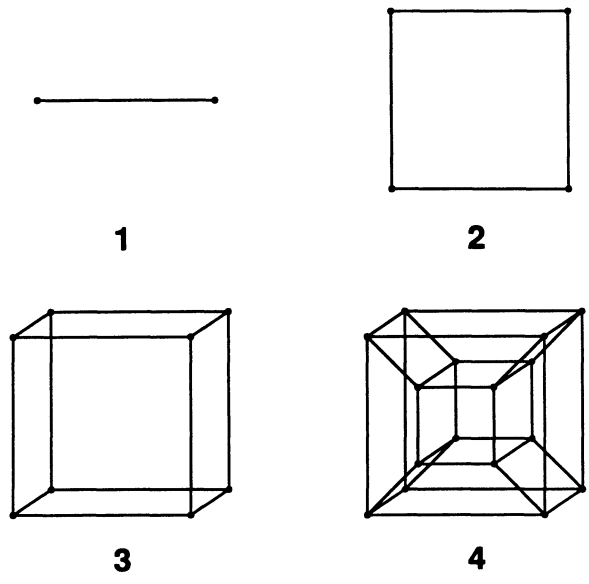


Figure 1. Illustration of hypercubes of dimensions 1, 2, 3, and 4. Vertices represent processors and edges represent direct interprocessor communication channels.

as 512 Kbytes of RAM. The host is an identical processing element with 2 Mbytes of RAM and access to a 20 Mbyte hard disk. The communication links are ethernet channels with a maximum bandwidth of 10 megabits per second.

A series of benchmark calculations were run on the Hypercube to determine its overall performance. Note that these benchmarks relate to the IPSC-1. The newer models of the IPSC have significantly faster processing and communication speeds. These indicate that if all 32 processors are running at 100% efficiency, the Hypercube operates at 0.62 million floating point operations per second. The node-to-node communication rate is 225 kbits per second for nearest-neighbor nodes. Although it is possible to send messages between non-nearest-neighbor nodes, this causes a significant degradation in communication speed. Taken together, these results indicate that a node can carry out 65 floating point operations in the time it can transmit 1 kbyte of data to the neighboring processor. The details of these benchmark results are given in Appendix A.

1.5. General Principles for Parallel Algorithms

Before discussing the quantum chemical algorithms it would be instructive to outline some general principles for programming distributed

memory computers and to describe a few of the details involved in programming the Hypercube. When designing an algorithm for a single-processor computer, the key to efficiency is to reduce the total number of operations. In contrast, for parallel algorithms the goal is to divide the problem into equal-sized tasks that can run concurrently. Unfortunately these two goals are often conflicting so that the best serial algorithm for a given problem is not necessarily the best starting point for an efficient parallel algorithm.[36] This means that it will often not be adequate simply to modify existing programs. Instead, the problem should be carefully reconsidered with special attention to how the task can be subdivided.

When dividing a problem into subtasks a number of constraints must be kept in mind. Most importantly, each subtask must be nearly the same computational size. That is, the computational load on each processor should be evenly balanced. Another consideration is how much communication will occur between processors. Obviously some communication must occur if a useful task is to be carried out, but the time the processor spends communicating is wasted and will decrease the performance relative to a single processor algorithm. An additional consideration is that communication should be carried out in an orderly, organized manner. This is to avoid having one communication channel getting overloaded (thereby becoming a bottleneck) and to ensure against the more serious problem of processor deadlock. Deadlock is the situation where a ring of processors becomes stuck because each processor in the ring is waiting for a message from the previous processor.

Given these constraints, what is the best strategy for dividing a problem into concurrent tasks? One alternative would be to write a unique program for each individual processing element. However, this approach is unsatisfactory for a number of reasons. One problem is that such an implementation will only work on a fixed number of processors, so that the performance could not be improved by adding more processing elements to the computer. Moreover, it would be prohibitively difficult to write separate programs for each of the tens or hundreds of processors available in most distributed memory computers.

A better strategy for subdividing a problem is to have one "control program" (usually running on a separate "host" processor) coordinating the activity of a single "slave" program, which runs on all of the remaining processors. This allows the use of a flexible number of processors since at run time the host program can read in the number of available processing elements and adjust the load on each processor accordingly. Also, this requires the writing of only two separate programs.

The first step in this strategy is to find one or more "dimensions" along which the problem can be divided. These dimensions are usually an adjustable parameter of the calculation, such as the number of particles in

a molecular dynamics problem, the number of basis functions in a quantum chemistry calculation, or the order of the matrices in a linear algebra computation. The subdivision of the task onto the processors then occurs along these dimensions. For example, if the problem involves the construction of a large matrix where each matrix element is the result of the same operations on different data sets, then the problem naturally divides up with each processor computing a equal sized sub-block of the matrix (see Figure 2).

Since most problems have a number of such dimensions, careful thought should be given to which are subdivided. The choice of dimensions requires consideration of both constraints dictated by the problem, such as load balancing and communications costs, and constraints dictated by the computer, such as the number and connectivity of the processors. For example, consider the simulation of the motion of a set of mutually interacting particles. The problem can be divided so that each processor is assigned either to propagate a fixed set of particles (regardless of where they are located) or to propagate all particles in a given region of space (regardless of how many particles it contains). The latter strategy is best for cases with a relatively uniform particle density—so that the computational load is balanced, and for cases with short-ranged interparticle forces—so

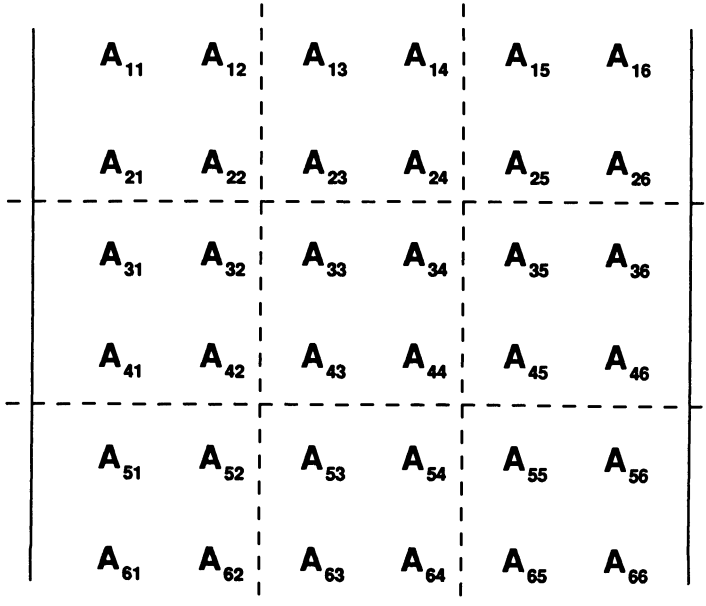


Figure 2. Distribution of matrix elements onto a 3×3 grid of processors. Dashed lines indicate processor boundaries.

that communication between processors is minimized. In order to avoid communication problems, however, this strategy requires a processor connectivity that supports a regular three-dimensional mesh.

As can be seen from this example there is no simple recipe for devising an efficient parallel algorithm—each problem requires careful consideration of constraints described above. In the following sections an effort has been made to clarify the steps in developing an efficient parallel algorithm by providing a step-by-step description of the development of these algorithms.

1.6. Programming the Hypercube

At the present time programming the Hypercube is much like programming a serial computer. The programs for both the host and node processors are written in either FORTRAN or C (all programs described in this chapter were written in FORTRAN). The interprocessor communication is facilitated by a set of routines that handle all of the communication protocol. Most of the intermode communication is carried out by with four routines: *send* and *recv* for asynchronous communication and *sendw* and *recvw* for synchronous communication. These routines take the same set of arguments: a channel number, a message type, the starting address and length of the message, and the destination node (or the sending node in the case of *recv* or *recvw*). The message type is a user-specified integer, which identifies the message. A message will be received by a processor only if its type matches that specified in the *recv* or *recvw* call.

The synchronous communication routines (*sendw* and *recvw*) block the operation of the program until the message has been received in the case of *recvw* or copied out of the user space in the case of *sendw*. The asynchronous routine *recv* simply queues a request for the receipt of a message. Similarly the *send* routine simply initiates the transmission of a message before returning to the program. The host communicates to the node processors via two synchronous communication routines: *sendmsg* and *recvmsg*. Although the routines described here are specific to the hypercube, they are sufficiently simple and general that similar routines will probably be standard on future distributed memory computers.

The use of these routines is illustrated by the simple Hypercube program in Figure 3. The host program reads in the number of processors to be used (*nproc*) and an array of numbers (x) equal in length to the number of processors. The host then sends out each of these numbers to a separate processor. The nodes receive the number, square it, and send it back to the host, where the numbers from all the processors are summed together (*total*).

a

```
      Program Host
C
C This program reads in a list of numbers, sends them
C to separate nodes for processing, and then sums the
C results together
C
      Integer*4 ci,mtype,length,node,pid,rlength
      Dimension x(32)
C
C Open communication channel
      ci=copen(1)
C
C Read in the number of node processors
      Read *,nproc
C
C Read in nproc numbers
      Read *,(x(i),i=1,nproc)
C
C Send these values to separate processors
      mtype=1
      length=4
      pid=1
      Do 10 i=1,nproc
      node=i-1
      Call sendmsg(ci,mtype,x(i),length,node,pid)
   10 Continue
C
C Receive values back from nodes
      total=0.0
      Do 20 i=1,nproc
      Call recvmsg(ci,mtype,value,length,rlength,node,pid)
      total=total+value
   20 Continue
C
C Print out result
      Print *,'Total =',total
      End
```

Figure 3. (a) Listing of sample program for the host processor. (b) Listing of sample program for the node processors.

b
```
      Program Node
C
C This program receives a value from the host, squares it
C and sends it back to the host.
C
      Integer*4 host,pid,mtype,length,rlength,ci,copen
C
C Open communications channel
      ci=copen(1)
C
C Receive value from host
      mtype=1
      length=4
      Call recvw(ci,mtype,value,length,rlength,host,pid)
C
C Square this number
      value=value*value
C
C Send new value back to host
      Call sendw(ci,mtype,value,length,host,pid)
C
      End
```

Figure 3. (*Continued*)

Finally, a few hardware features unique to the Hypercube should be considered since they will affect the design of the programs to be discussed. Most important of these is the limited memory available on the processing nodes. Although there are 512 kbytes (thousand bytes) of memory available on each node, the node operating system requires nearly 200 kbytes, leaving only 320 kbytes of usable memory. Since the nodes have no access to mass storage and there is no provision for program overlay, all of the programs and data for a given calculation must fit into this limited space. This limitation will be discussed further when the benchmarks are presented.

A second item requiring special consideration is the Hypercube's very low ratio of processing speed to communication speed (see Appendix A). In its standard configuration, the Hypercube can send a word (8 bytes) of data in nearly half the time it takes to carry out a floating point operation. This will clearly not be the case for future distributed memory computers, and a vector processing board is now available for the Hypercube that boosts

the processing speed by a factor of 100 (without increasing the communication speed). For these reasons the algorithms were designed assuming a relatively high processing speed to communication speed ratio.

2. Parallel Algorithms for Molecular Quantum Mechanics

2.1. Integral Evaluation

An important consideration in developing a package of quantum chemistry programs is that most calculations involve running a series of these programs in sequence (for example, an integral evaluation followed by an SCF calculation). Hence, in order to optimize the efficiency of the overall computation it is necessary to design the set of programs to facilitate the most computationally demanding steps. This can be complicated, because it often means choosing less than optimal algorithms for the simpler steps. The computational complexities of a number of common quantum chemical procedures are given in Table 1.

The first step in most quantum chemical calculations is the numerical evaluation of various integrals. There are three sets of one-electron (two-index) integrals: the nuclear repulsion, kinetic energy, and overlap integrals. Each of these sets contains $[n^*(n+1)]/2$ integrals, where n is the number of basis functions. The remaining integrals are the two-electron (four-index) integrals [equation (4)]. This set is much larger than the one-electron integral sets, containing $n^4/8$ symmetry distinct integrals. Since the calculation of the two-electron integrals vastly dominates the integral evaluation step, the emphasis should be to parallelize this step. The computation of the one-electron integrals will be carried out on the host processor.

In principle the parallelization of the two-electron integrals is straightforward; each processor computes an equal-sized subset of the total

Table 1. Computational Complexities

Procedure	Complexity[a]
Integral evaluation	$O(n^4)$
SCF	$O(n^4)$
Integration transformation	$O(n^5)$
MPPT (second order)	$O(n^4)$
MPPT (third order)	$O(n^6)$
MPPT (fourth order)	$O(n^7)$
CISD	$O(n^6)$

[a] n, Number of basis functions.

integral list. However, the distribution of the integrals onto the processor is complicated by considerations of how the integrals will be used in subsequent computational steps. In the development of this integral program two common sequences of quantum chemical computational procedures were considered. The first of these (the most common of all quantum chemical computations) is simply an SCF energy calculation. The second sequence is an SCF calculation followed by a post-SCF procedure requiring an integral transformation.

2.1.1. Integral Distribution for SCF Calculations

The computational complexity of the integral evaluation and SCF steps are the same, so neither step should be given dominant consideration. Further, since an efficient SCF algorithm can be designed that will work with arbitrary integral distributions, it is necessary only to focus on efficient load balancing of the integral evaluation. The simplest method to distribute equally the integrals is to have each processor loop over all symmetry distinct integrals and select a unique subset to calculate and store. This can be accomplished by the segment of program given below:

```
ICOUNT = 0
JCOUNT = 0
DO μ = 1, NBASIS
  DO ν = 1, μ
    DO λ = 1, μ
      DO σ = 1, λ (ν if μ = ν)
        ICOUNT = ICOUNT + 1
        IF ((MOD(ICOUNT, NUMPROC)) .EQ. PROC_ID) THEN
        EVALUATE INTEGRAL (μν|λσ)
      END DO (σ)
    END DO (λ)
  END DO (ν)
END DO (μ)
```

Each processing node has a unique identification number (*PROC_ID*) ranging from 0 to the total number of processors minus 1. An integral is evaluated if the integral's cardinality (*ICOUNT*) modula the total number of processors (*NUMPROC*) is equal to the node's identification number.

This scheme guarantees equal balancing of the total number of integrals on each processor. A difficulty arises, however, since some basis functions represent more complex atomic orbitals than others (*p* or *d* orbitals as opposed to *s* orbitals), so that the computational cost for evaluating each integral is not the same. This means that even though each processor

has an equal number of integrals, the net computational load may not be evenly balanced. Unfortunately a more exact load balancing would be very difficult to achieve. Even if the exact computational cost of each integral type were known in advance, equal distribution of the load would be an extremely difficult computational task. (This task is equivalent to the "multidimensional knapsack" problem, which is in the class of NP complete problems widely believed to have an exponential computational complexity).[37]

A possible alternative is to use dynamic load balancing. In this strategy a certain number of integrals are held back and then are dealt out to the first node processors to finish their initially assigned integrals. The decision was made not to implement dynamic load balancing in the integral evaluation step since it would greatly increase the complexity of the program and take up much-needed memory space. Moreover, the simple scheme proposed evenly distributes both the computationally simple

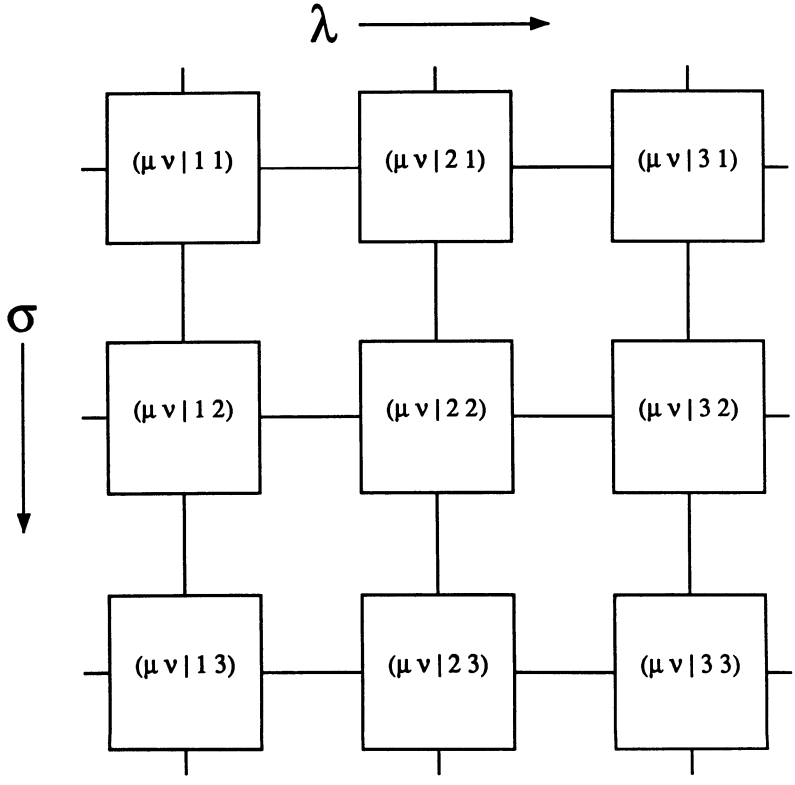


Figure 4. Distribution of two electron integrals for $n = 3$ on 3×3 grid of processors. The Greek indices range over all symmetry distinct values: $\mu, \nu = 1\text{–}n$, with $\mu \geqslant \nu$.

and complex integrals so that for large problem sizes the load balancing should be quite good. (This is confirmed by benchmark results given in the following section.)

2.1.2. *Integral Distribution for Post-SCF Calculations*

If the two-electron integrals are eventually to be transformed, a number of complicated constraints are placed on the distribution scheme. The transformation is simplified if the integrals are available in continuous segments (rather than scattered as in the method previously described). Moreover, the communication costs in the transformation step are greatly reduced if some integral redundancy is allowed.

For reasons that will be explained in the description of the transformation step, a good way to distribute the integrals is to map them onto a two-dimensional rectangular array of processors (a subtopology of the

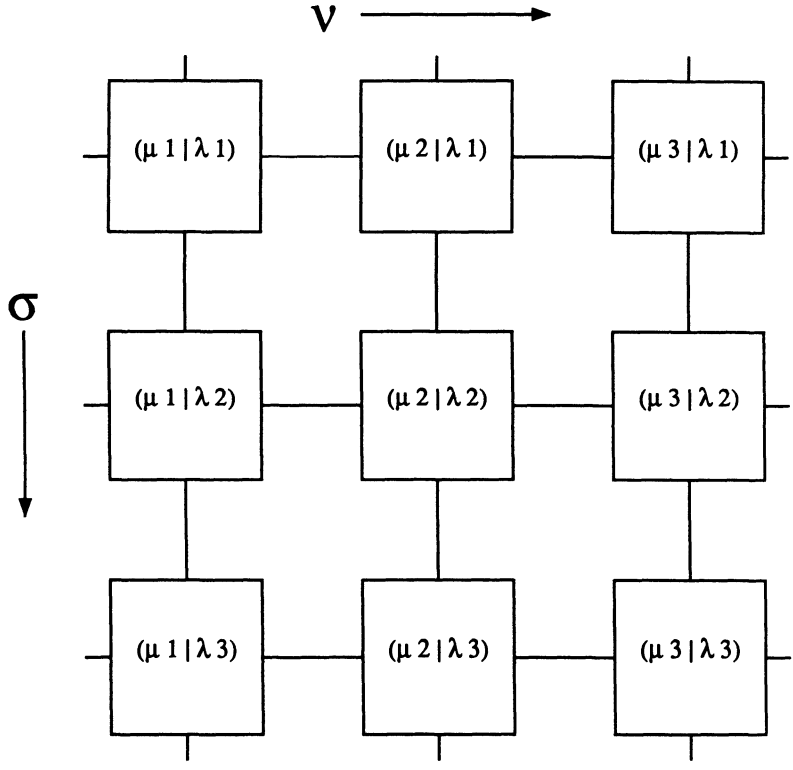


Figure 5. Distribution of two electron integrals for $n = 3$ on 3×3 grid of processors. The Greek indices range over all values: $\mu, \lambda = 1\text{–}n$.

Hypercube connectivity). A simple way to carry out this mapping is to choose two of the integral's four indices as the coordinates of the processing node that is to evaluate the integral (Figures 4–7). If there are more basis functions than processors along one of the dimensions, then more than one basis function per processor is assigned along that dimension (see Figures 6 and 7).

Because of the inherent symmetry of the two electron integrals, there are only two symmetry distinct pairs of indices that can be chosen: the last two indices λ, σ, and the second and fourth ν, σ. The choice of which two indices to use for the integral distribution is determined by what will be done with the transformed integrals. Figures 4–7 illustrate these two possibilities, both for the case where the number of basis functions matches the number of processors in each dimension (Figures 4 and 5) and the case where the number of basis functions exceeds the number of processors along each dimension (Figures 6 and 7).

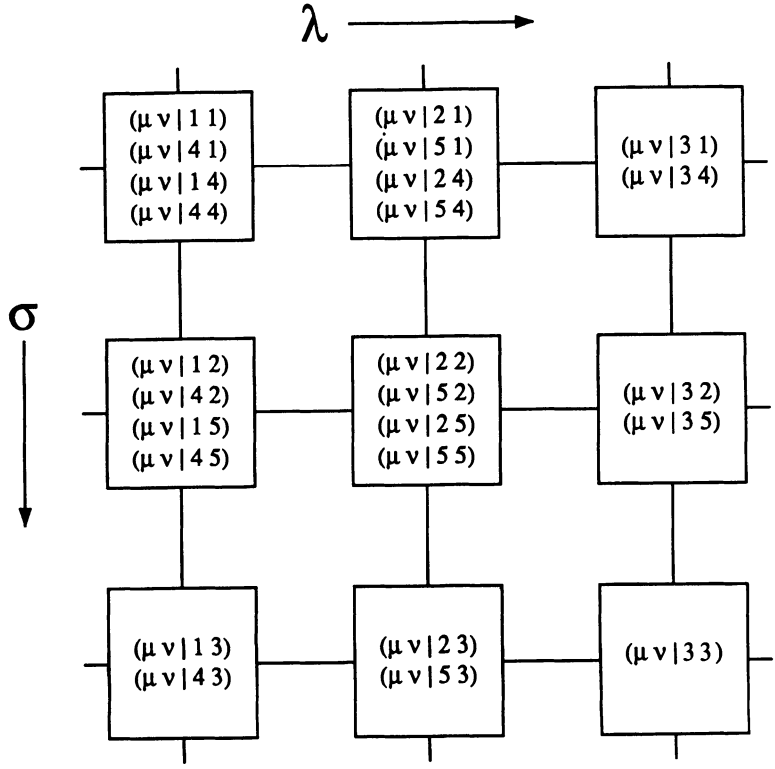


Figure 6. Distribution of two electron integrals for $n = 5$ on 3×3 grid of processors. The Greek indices range over all symmetry distinct values: μ, $\nu = 1$–n, with $\mu \geqslant \nu$.

These two distribution schemes involve the calculation of different numbers of integrals. The first scheme easily allows the use of one of the three index symmetries (permutation of the first two indices) so that there is a fourfold redundancy in the integrals calculated. The second scheme does not allow the easy use of any of the integral symmetries so that all N^4 integrals are calculated. Of course the calculation of redundant integrals means that the integral evaluation step is nonoptimal; however, this integral distribution will facilitate the more computationally complex integral transformation and post-SCF steps. Both of these distribution schemes will give good load balancing for sufficiently large problem sizes.

2.1.3. *Two-Electron Integral Evaluation*

Methods for the numerical evaluation of two-electron integrals have been the focus of intense research efforts for the past three decades[38] and are still a subject of much interest.[39] The result of these efforts have been

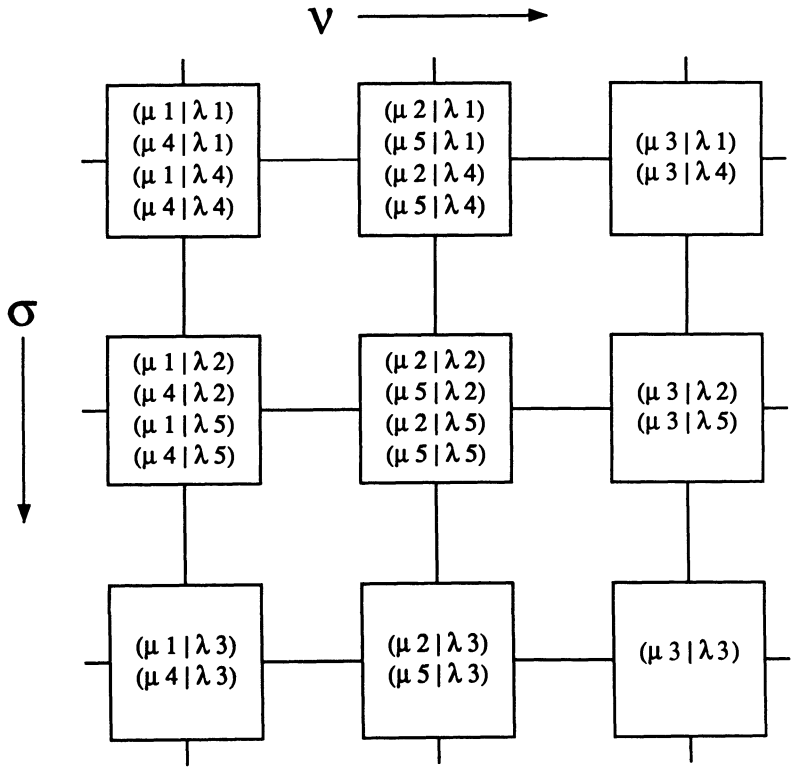


Figure 7. Distribution of two electron integrals for $n = 5$ on 3×3 grid of processors. The Greek indices range over all values: $\mu, \lambda = 1\text{–}n$.

a large number of computationally efficient methods for integral evaluation. Although the integral evaluation scheme chosen for the Hypercube was dictated by memory considerations, the question of which methods are optimal for parallelization should be addressed.

In the most efficient integral evaluation schemes two strategies are used to improve performance. One is the precomputation of large tables of repeatedly used numerical values. The other strategy is to compute the integrals in large groups to eliminate the redundant computation of partial terms. Both of these present special difficulties for parallel implementations which must be carefully considered.

A potential problem with the precomputation of lookup tables is that it does not use memory very efficiently since each of the processing nodes would have to store a copy of the table. Another concern is the manner in which the table is computed. It is obviously undesirable to have each processor calculate the entire table; however, if each processor computed unique segments, a large amount of communication would be required to fully distribute the table.

The most common approach to the second strategy is to calculate the integrals in complete shell blocks. A shell block is the set of all integrals over common atomic shells. For example, a complete $(\phi_p\phi_s|\phi_s\phi_s)$ block contains the following three integrals: $(\phi_{p_x}\phi_s|\phi_s\phi_s)$, $(\phi_{p_y}\phi_s|\phi_s\phi_s)$, and $(\phi_{p_z}\phi_s|\phi_s\phi_s)$. Table 2 lists the sizes of other shell blocks.

The rationale for this strategy is that the mathematical expressions for each integral in the shell blocks contain many common parts, so that if the entire block is calculated at once, many potentially redundant steps are avoided. This approach has proved to be very efficient on serial computers, but a number of problems arise when attempting to construct a parallel version of this scheme. These problems stem from the fact that to be efficient, the computation of the integrals in an individual shell block cannot be split across processors. This leads to the possibility of very poor load balancing due to the large discrepancies in the sizes of the shell blocks.

Table 2.
Shell Block Sizes

Shell block	Size
$(s\,s\,|\,s\,s)$	1
$(p\,s\,|\,s\,s)$	3
$(p\,p\,|\,s\,s)$	9
$(p\,s\,|\,p\,s)$	9
$(p\,p\,|\,p\,s)$	27
$(p\,p\,|\,p\,p)$	81

A further difficulty that arises if the integrals are distributed by shell block is that the resulting distribution will complicate subsequent integral transformation and post-SCF steps. Despite the difficulties cited here, these two strategies potentially offer very large increases in the efficiency of the integral evaluation step, so despite their drawbacks, these methods should be carefully considered in future implementations of parallel integral evaluation programs.

As mentioned earlier, the prime concern in developing an integral program for the Hypercube was to conserve memory (since a fast integral evaluator would be of little use if there was no memory remaining to store the integrals.) Hence the decision was made to write a new integral evaluation program rather than use one of the available programs. The expressions used for both the one- and two-electron integrals are those derived by Taketa, Huzinaga, and O-ohata.[40] These expressions have the form of sums of products involving numerical constants, input values (angular momenta, internuclear distances), and the so-called associated F-function:

$$F_m(t) = \int_0^1 t^{2m} \exp(-xt^2)\, dt \tag{11}$$

An advantage of this method over the more efficient techniques is that the expressions are completely general so that integrals can be calculated over basis functions of arbitrarily high angular momentum.

In the implementation on the Hypercube, each integral is evaluated individually, so that no efficiency is gained by avoiding redundant computation. The overall size of the two-electron integral program based on this method is 55000 bytes. In comparison, a standard two-electron integral package (excluding lookup tables) requires nearly 800,000 bytes. For the simplest two-electron integrals, $(\phi_s\phi_s|\phi_s\phi_s)$, the Hypercube implementation is about a quarter the speed of the larger program.

2.2. Self-Consistent Field Calculation

The SCF procedure is more complex and less homogeneous than the integral evaluation and hence a much more difficult task to parallelize. The first step is to look in detail at the various steps involved and the associated computationally complexity. The steps involved in an SCF iteration for a closed shell system are listed below:

1. Generate initial guess for the density matrix.
2. Form two-electron part of the Fock matrix:

$$F_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu|\lambda\sigma) - \tfrac{1}{2}(\mu\lambda|\sigma\nu)] \tag{12}$$

3. Add kinetic and potential energy one-electron integrals:

$$F_{\mu\nu} = F_{\mu\nu} + K_{\mu\nu} + V_{\mu\nu} \tag{13}$$

4. Transform the Fock matrix using the overlap integrals:

$$\mathbf{F}^{\tau} = \mathbf{S}^{-1/2}\mathbf{F}\mathbf{S}^{-1/2} \tag{14}$$

5. Diagonalize the transformed Fock matrix to get the SCF vector **C** and the orbital energies ε:

$$\mathbf{F}^{\tau}\mathbf{C}^{\tau} = \varepsilon\mathbf{C}^{\tau} \tag{15}$$

6. Back transform the SCF vector:

$$\mathbf{C} = \mathbf{S}^{-1/2}\mathbf{C}^{\tau} \tag{16}$$

7. Form new density matrix (NOCC, number of occupied orbitals):

$$P_{\mu\nu} = 2\sum_{i}^{\mathrm{NOCC}} C_{\mu i}C_{\nu i} \tag{17}$$

Steps 2–7 are repeated until the density matrix is converged. The computational complexity associated with each of these steps is given in Table 3. It is obvious from the data in Table 3 that for large problem sizes the SCF computation will be dominated by step 2, the formation of the two-electron portion of the Fock matrix. This result is corroborated by actual timings on serial SCF programs.

Hence, the logical starting point in the parallelization of the SCF procedure is to distribute the computation of the two electron Fock matrix. The two components required to form this term are the density matrix and the two-electron integrals. Since the two-electron integrals have already

Table 3.
Computation Complexities

Step	Complexity
1	$O(n^2)$
2	$O(n^4)$
3	$O(n^2)$
4	$O(n^3)$
5	$O(n^3)$
6	$O(n^3)$
7	$O(n^2)$

been distributed onto the nodes by the integral program and these integrals do not change with each iteration, a sensible procedure for forming the Fock matrix is to distribute the density matrix to the nodes each iteration and then calculate partial Fock matrices from the resident two-electron integrals. These partial Fock matrices are then summed together on the host processor where the new density matrix is formed and broadcast back to the nodes. A flow diagram of this procedure is given in Figure 8.

The method used to form the partial two-electron Fock matrices is dependent on the integral distribution scheme used in the integral evaluation step. If the integrals were distributed for just an SCF calculation the formation of the partial Fock matrices is straightforward since a nonredundant list of two-electron integrals is evenly distributed onto the processors. This procedure on each node involves simply looping over all of the resident integrals calculating their contributions to the Fock matrix. Since only the symmetry distinct integrals have been stored, each integral will contribute to between one and eight Fock matrix elements. Figure 9 lists all Fock matrix element contributions from symmetry distinct integral types.

If an integral transformation is to be carried out, the formation of the partial Fock matrices is more difficult because of the more complex integral distribution. The difficulty is that redundant integrals are stored so that special care must be taken that redundant contributions are not made to the partial Fock matrices. This consideration greatly complicates load balancing. Since the integral distribution is determined by two of the integral indices, most schemes for selecting symmetry distinct integrals will preferentially load certain processors. However, a scheme that works well for most cases is straightforward. Consider the simplest integral loading (as in Figure 4) where a processor with coordinates k, l holds the integral block $(ij|kl)$ where $i, j = 1\text{–}n$. If $k < l$ the node processes all symmetry distinct integrals for which $i+j$ is odd. If $k > l$ then the node processes all symmetry distinct integrals for which $i+j$ is even. Of course if $k = l$ then all of the symmetry distinct integrals must be processed. Hence this strategy will overload the diagonal processors ($k = l$), but since for most actual cases each node will have many integral blocks (as in Figures 6 and 7) this load imbalance should not be serious.

As indicated in the flow diagram (Figure 8), the SCF procedure requires the communication of all of the partial Fock matrices from the nodes to the host and the subsequent rebroadcast of the new density matrices. If all of the communication were to be carried out directly between the nodes and the host, the host's communication channels would be overloaded and become a bottleneck. What is needed is to carry out as much of the communication in parallel as is possible and to avoid having an individual processor sending or receiving more than one message at a time. Fortunately there is a simple scheme utilizing a so-called broadcast

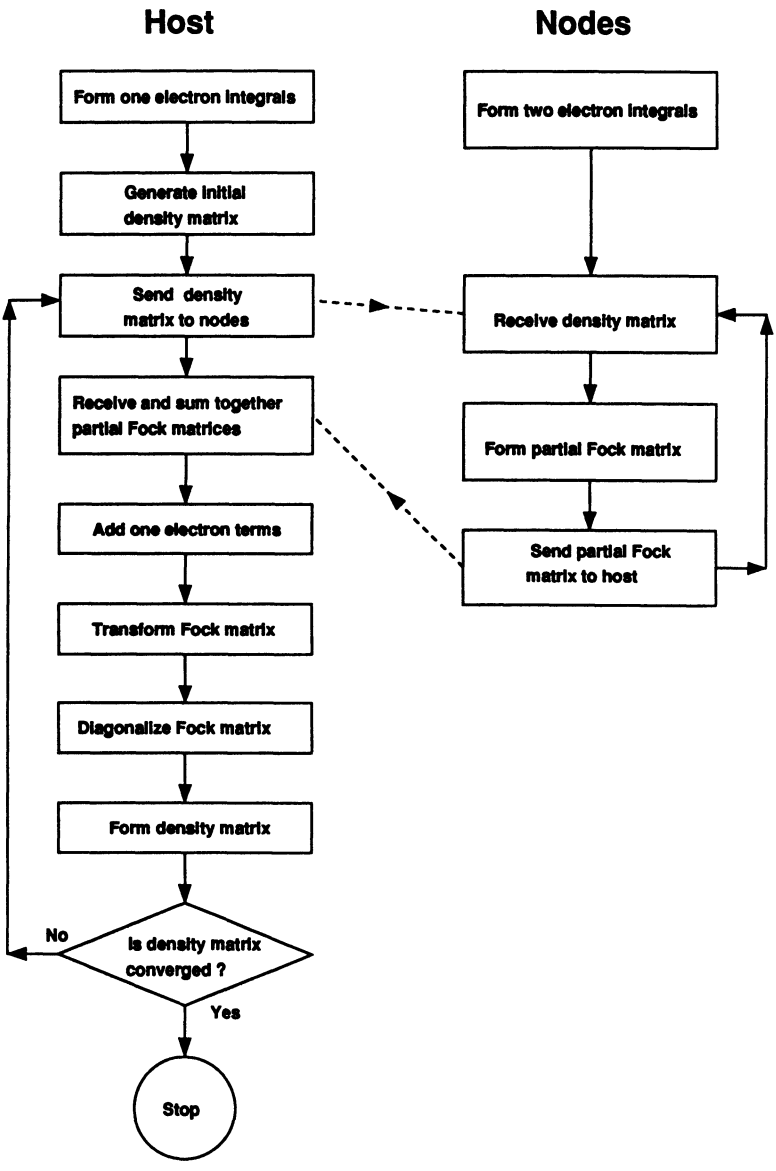


Figure 8. Flow diagram of integral and SCF programs. Dashed lines indicate interprocessor communication.

Fock Matrix Contributions									
Integral	F_{aa}	F_{ba}	F_{bb}	F_{ca}	F_{cb}	F_{cc}	F_{da}	F_{db}	F_{dc}
$(aa \mid aa)$	$\frac{1}{2}P_{aa}$	-	-	-	-	-	-	-	-
$(ba \mid aa)$	P_{ba}	$\frac{1}{2}P_{aa}$	-	-	-	-	-	-	-
$(bb \mid aa)$	P_{bb}	$\frac{-1}{2}P_{ba}$	P_{aa}	-	-	-	-	-	-
$(ba \mid ba)$	$\frac{-1}{2}P_{bb}$	$\frac{3}{2}P_{ba}$	$\frac{-1}{2}P_{aa}$	-	-	-	-	-	-
$(bb \mid ba)$	-	$\frac{1}{2}P_{bb}$	P_{ba}	-	-	-	-	-	-
$(cb \mid aa)$	$2P_{cb}$	$\frac{-1}{2}P_{ca}$	-	$\frac{-1}{2}P_{ba}$	P_{aa}	-	-	-	-
$(ca \mid ba)$	$-P_{cb}$	$\frac{3}{2}P_{ca}$	-	$\frac{3}{2}P_{ba}$	$\frac{-1}{2}P_{aa}$	-	-	-	-
$(cb \mid ba)$	-	$\frac{3}{2}P_{cb}$	$-P_{ca}$	$\frac{-1}{2}P_{bb}$	$\frac{3}{2}P_{aa}$	-	-	-	-
$(ca \mid bb)$	-	$\frac{-1}{2}P_{cb}$	$2P_{ca}$	P_{bb}	$\frac{-1}{2}P_{aa}$	-	-	-	-
$(cc \mid ba)$	-	P_{cc}	-	$\frac{-1}{2}P_{cb}$	$\frac{-1}{2}P_{ca}$	$2P_{ba}$	-	-	-
$(cb \mid ca)$	-	$\frac{-1}{2}P_{cc}$	-	$\frac{3}{2}P_{cb}$	$\frac{3}{2}P_{ca}$	$-P_{ba}$	-	-	-
$(dc \mid ba)$	-	$2P_{dc}$	-	$\frac{-1}{2}P_{db}$	$\frac{-1}{2}P_{da}$	-	$\frac{-1}{2}P_{cb}$	$\frac{-1}{2}P_{ca}$	$2P_{ba}$
$(db \mid ca)$	-	$\frac{-1}{2}P_{dc}$	-	$2P_{db}$	$\frac{-1}{2}P_{da}$	-	$\frac{-1}{2}P_{cb}$	$2P_{ca}$	$\frac{-1}{2}P_{ba}$
$(da \mid cb)$	-	$\frac{-1}{2}P_{dc}$	-	$\frac{-1}{2}P_{db}$	$2P_{da}$	-	$2P_{cb}$	$\frac{-1}{2}P_{ca}$	$\frac{-1}{2}P_{ba}$

Figure 9. Symmetry distinct two-electron integral contributions to the Fock matrix. For example, the integral $(21 \mid 11)$ has contributions to two Fock matrix elements: $P_{21} * (21 \mid 11)$ to F_{11} and $\frac{1}{2}P_{11} * (21 \mid 11)$ to F_{21}.

tree that fulfills these conditions. The broadcast of a message from the host to all of the nodes in a three-dimensional hypercube is shown in Figure 10.

Basically, the scheme involves the sequential broadcast of the message from all of the nodes in a N-dimensional Hypercube to all the nodes in an $(N+1)$-dimensional Hypercube. Hence, it takes a total of six communication steps to broadcast a message to all of the nodes in a five-dimensional (32-processor) Hypercube. (This includes the initial communication of the message from the host to node 0.) Similarly, if a result is to summed from the nodes onto the host, the reverse procedure can be carried out: each node in an N-dimensional cube receives a message, sums it into his own result, and broadcasts this message to the corresponding processor in an $(N-1)$-dimensional cube.

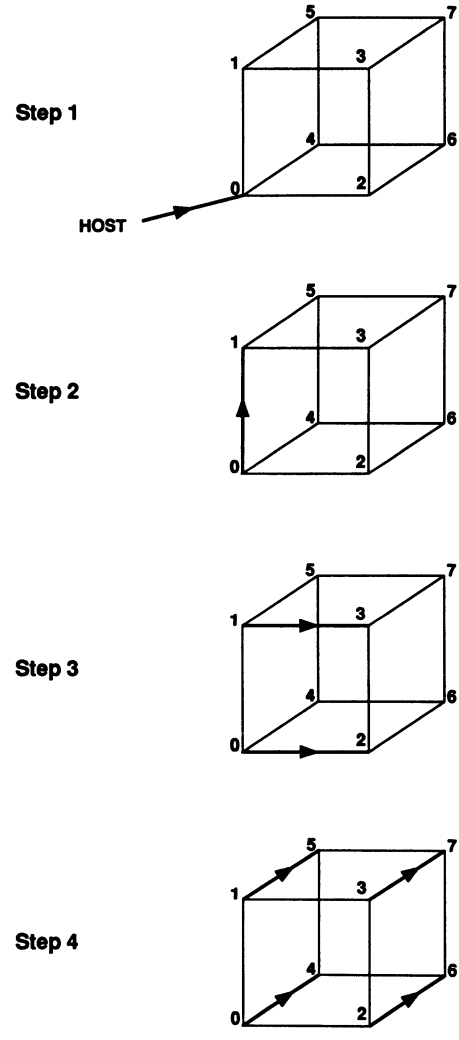


Figure 10. Communication of message from host to all nodes using broadcast tree.

Although parallelizing the most computationally complex part of the SCF procedure is clearly the first step, there are a number of reasons also to distribute the less computationally complex tasks. One obvious reason is that for sufficiently large problem sizes, even the $O(n^3)$ steps in the SCF procedure will take a prohibitively long time on the host processor. A second reason is that for problems running on computers where the number of processors (m) is comparable to the number of basis functions (n), the complexity of the Fock matrix formation $O(n^4/m)$ reduces to $O(n^3)$

so that the other $O(n^3)$ steps will dominate the computation time. (This is the case for some of the benchmark calculations on the Hypercube where the memory size severely limits the number of basis functions.)

There are two types of $O(n^3)$ steps occurring in the SCF procedure: the matrix multiplications and the diagonalization of the Fock matrix. The matrix multiplications are fairly easy to distribute, but the diagonalization is much more difficult to parallelize efficiently. Although a parallel diagonalization has not yet been implemented, the method for doing this is discussed here.

2.3. Parallel Matrix Diagonalization

Because of their tremendous importance in a vast number of mathematical and scientific applications, numerical methods for the computation of eigenvalues and eigenvectors of symmetric matrices have been the subject of intense study for more than century. A large number of very efficient serial algorithms have been developed which are able to compute the eigenvectors of an $n \times n$ matrix in $O(n^3)$ time.[41] Not surprisingly, in recent years a lot of effort has been put into the study of parallel matrix diagonalization algorithms. One conclusion of this work is that the very best serial algorithms (e.g., Householder tridiagonalization followed by QR iterations) are not as well suited for parallelization as the older, simpler methods.

One very old algorithm that seems especially promising for parallel applications is the Jacobi method (first described in 1846).[42] In this scheme a symmetric matrix A is diagonalized by a series of plane rotations. Each rotation is carried out such that it annihilates $n(n-1)/2$ of the off-diagonal matrix elements. Since the zeroed off diagonal matrix elements will not necessarily remain zero for successive rotations, the method is iterative. If m such rotations are necessary to zero all of the off diagonal elements of A, then we have

$$\mathsf{U}\mathsf{A}\mathsf{U}^{\dagger} = \mathsf{R}_m \mathsf{R}_{m-1} \cdots \mathsf{R}_1 \mathsf{A} \mathsf{R}_1^{\dagger} \mathsf{R}_2^{\dagger} \cdots \mathsf{R}_m^{\dagger} \tag{18}$$

where U is the matrix of eigenvectors of A and R_m is the mth successive plane rotation matrix. The eigenvalues are given by

$$\mathsf{U} = \mathsf{R}_m \mathsf{R}_{m-1} \cdots \mathsf{R}_1 \tag{19}$$

and the rotation matrix elements at each iteration are given by

$$R_{ii} = R_{jj} = \cos(\alpha_{ij}) \tag{20}$$

$$R_{ij} = -R_{ji} = \sin(\alpha_{ij}) \tag{21}$$

where

$$\tan(2\alpha_{ij}) = 2\frac{A_{ij}}{A_{ii} - A_{jj}} \tag{22}$$

Since the computation of the eigenvectors simply corresponds to a series of matrix multiplications, there are a number of possible strategies for parallelization of the Jacobi algorithm. One approach has been investigated by Sameh[43] and Whiteside *et al.*[44] This scheme is based on the observation that there exist sets of plane rotations that each annihilate different matrix elements without changing the values of the matrix elements annihilated by other rotations in the same set. Since all rotation matrices in each set commute with each other, the rotations can be carried out independently. For an $n \times n$ matrix there are $n/2$ rotation matrices in each of these sets so that $n/2$ off-diagonal elements could be eliminated simultaneously. Thus, this algorithm could efficiently utilize $n/2$ processors in parallel. For the SCF procedure described here, $\sim n^2$ processors are used to form a Fock matrix of order n. However, this diagonalization routine would only use $\sim n$ processors. Hence, this algorithm is poorly balanced for the SCF procedure.

Fortunately another parallel Jacobi diagonalization algorithm has recently been developed that allows the efficient use of much larger numbers of processors.[45] In this scheme the matrix is divided into 2×2 submatrices and mapped onto a two-dimensional square array of processors, so that an $n \times n$ matrix would map onto an $(n/2) \times (n/2)$ array of processors (see Figure 2). During each iteration the diagonal processors (i.e., those containing diagonal matrix elements) calculate the rotation matrix necessary to diagonalize their resident 2×2 matrix. This rotation matrix is broadcast to the other processors, which carry out the necessary rotations on the off-diagonal terms. After one such complete rotation has been carried out, adjacent processors exchange rows and columns so that the diagonal processors receive new off-diagonal matrix elements and the rotation is again carried out. The method has been empirically found to require $O(\log(n))$ such rotations, with each rotation requiring $O(n)$ computing time, yielding a net computational complexity of $O(n \log(n))$. Although this method has not yet been implemented on the Hypercube, its low computation cost, orderly interprocessor communication, and efficient use of $\sim n^2$ processors make it a good candidate for the Fock matrix diagonalization.

2.4. Two-Electron Integral Transformation

The two-electron integral transformation is the most computationally complex of the steps so far considered. Hence, much of the concern in the

design of the algorithms for the previous two steps was to facilitate the efficiency of this computation. This procedure involves the transformation of all four indices of the two-electron integrals. The most compact expression for this step is

$$(ij|kl) = \sum_{\mu}\sum_{\nu}\sum_{\lambda}\sum_{\sigma} C_{\mu i} C_{\nu j} C_{\lambda k} C_{\sigma l} (\mu\nu|\lambda\sigma)$$

where the Latin letters represent MO indices and the Greek letters AO indices. Since this $O(n^4)$ task must be carried out for all of the $n^4/8$ integrals, the total complexity as written above is $O(n^8)$. However, this transformation can be rewritten as a series of one index transformations (or "quarter transformations"). For example, the transformation of the fourth index, σ can be carried out as an autonomous step:

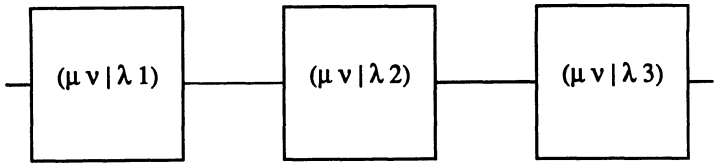
$$(\mu\nu|\lambda l) = \sum_{\sigma} C_{\sigma l} (\mu\nu|\lambda\sigma)$$

Each of these quarter transformations involves n^5 multiplications, so that the entire sequence of one-index transformations requires $4n^5$ multiplications. If the eightfold index symmetry is exploited, this total can be reduced to $\frac{5}{2}n^5$. More sophisticated transformations have been developed that involve the decomposition of each quarter transformation and require a total of only $(25/24)\,n^5$ multiplications.[46] However, as pointed out previously, the optimal serial algorithms are usually too complex to be converted into an efficient parallel implementation. Instead, it is usually better to begin with a simple expression for the problem and to look for dimensions along which to divide the task.

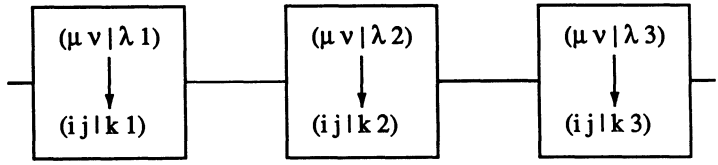
In the transformation step a natural choice for these dimensions would be one or more of the two-electron integral indices. The advantage of this choice is that all of the integrals needed to complete a given quarter transformation would be present on a single processor or along a single dimension of the processor array. To illustrate this concept consider the mapping of the two-electron AO integrals onto a one-dimensional array of processors (i.e., a loop). (A three basis function example is shown in Figure 11). The integral blocks are assigned to processors on the basis of their fourth index, σ. Each integral block contains integrals for all symmetry distinct combinations of μ, ν, and λ: $\mu, \nu = 1\text{–}n$ with $\mu \geqslant \nu$, and $\lambda = 1\text{–}n$. Note that only one of the three axes of integral symmetry is used, so that a fourfold redundancy of integrals is stored.

For this integral distribution, the first step is the transformation of the indices local to each processor, μ, ν, and λ (step 2 in Figure 11). For example, on the first processor this involves

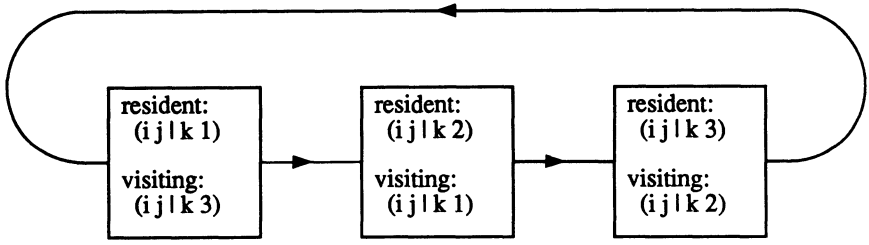
$$(ij|k1) = \sum_{\mu}\sum_{\nu}\sum_{\lambda} C_{\mu i} C_{\nu j} C_{\lambda k} (\mu\nu|\lambda 1)$$



Step 1: Initial distribution of AO integrals.



Step 2: In-place transformation of first 3 indices.



Step 3: Transformation of final index requiring communication around processor ring.

Figure 11. Transformation of three basis function case mapped onto a one dimensional grid of three processors.

(Of course in an actual implementation this would be carried out as three quarter transformations.) Because of the integral distribution this step can be carried out without any interprocessor communication.

All that remains is the transformation over the final index, σ. Since each processor holds integral blocks with only a single value of σ and this final quarter transformation requires a summation over this index, this step will require interprocessor communication. Because of the distribution scheme, this can be carried out in a series of orderly communication steps as follows (step 3 in Figure 11). First, each processor sends a copy of the

partially transformed integrals to its neighboring processor. Each processor then sums a contribution from the visiting integral block into a buffer:

$$(ij|kl_{\text{res}}) = (ij|kl_{\text{res}}) + C^{l_{\text{res}}}_{\sigma_{\text{vis}}}(ij|k\sigma_{\text{vis}})$$

where the subscripts res and vis indicate indices associated with the resident and visiting integral blocks. Note the assumption that the processors will retain the same AO and MO integral blocks: $l_{\text{res}} = \sigma_{\text{res}}$.

This step is repeated until the integral blocks have been passed completely around the loop and have arrived at their initial nodes. At this point the transformation is complete since each processor has received all the necessary contributions for the last quarter transform.

Obviously this one-dimensional mapping could be extended to two, three, or four dimensions by simply distributing more of the integral indices in the way σ was distributed in the one-dimensional case. The choice of how many dimensions to use is essentially that of the granularity of the parallelism.

The granularity of the parallelism refers to the amount of independent computation that each processor performs before it must communicate with another processor. If a large amount of processing is performed, for example a pass through some outer loop of the program, then the granularity is "coarse." If only a small amount of processing is performed between communication steps, for instance only a single arithmetic operation, then the parallelism is "fine grained." Because of the relatively slow interprocessor communication channels on the Hypercube (probably this will be a problem on all distributed memory machines), fairly coarse-grained communication is favored. Otherwise, the communication overhead will be much more time consuming than the useful computations performed.

Other considerations in the choice of dimensionality are the number of available processing nodes and the total amount of memory on each node. If the integrals are mapped onto a one-dimensional grid (see Figure 11), the largest number of processors that can be utilized is equal to the number of basis functions, n, and each node must store $\sim n^3$ integrals. Similarly, for a two-dimensional mapping, the maximum number of usable nodes would be n^2, with each node storing $\sim n^2$ integrals. Table 4 lists the relevant data for each of the possible mappings.

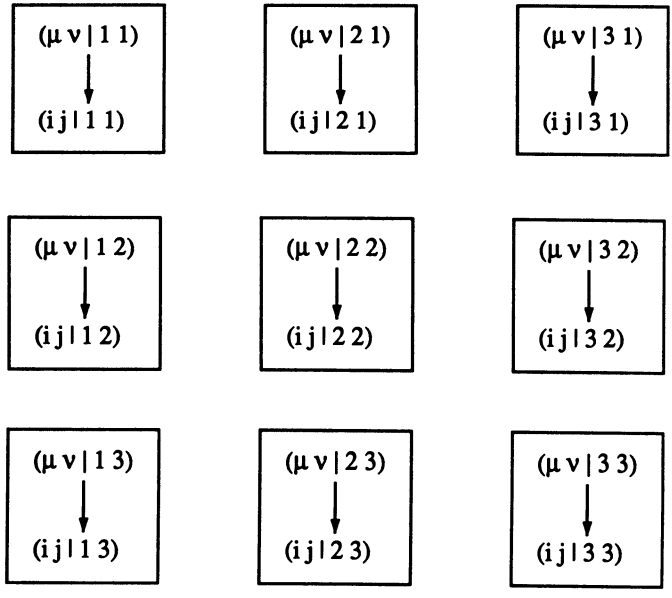
For most chemically interesting problems, the number of basis functions is not more than a few hundred. In comparison, the Intel Hypercube is available with up to 128 processors and other computers are available with thousands of processors. Hence, the one-dimensional transformation scheme would not be able to utilize fully the available hardware. However, the two-dimensional mapping would be able to use efficiently the large

Table 4. Integral Distributions

Dimensions	No. of usable nodes	Integrals per node	Operations per communication
1	n	n^3	n^3
2	n^2	n^2	n^2
3	n^3	n	n
4	n^4	1	1

parallel computers available in the foreseeable future while still maintaining a fairly coarse-grained parallelism.

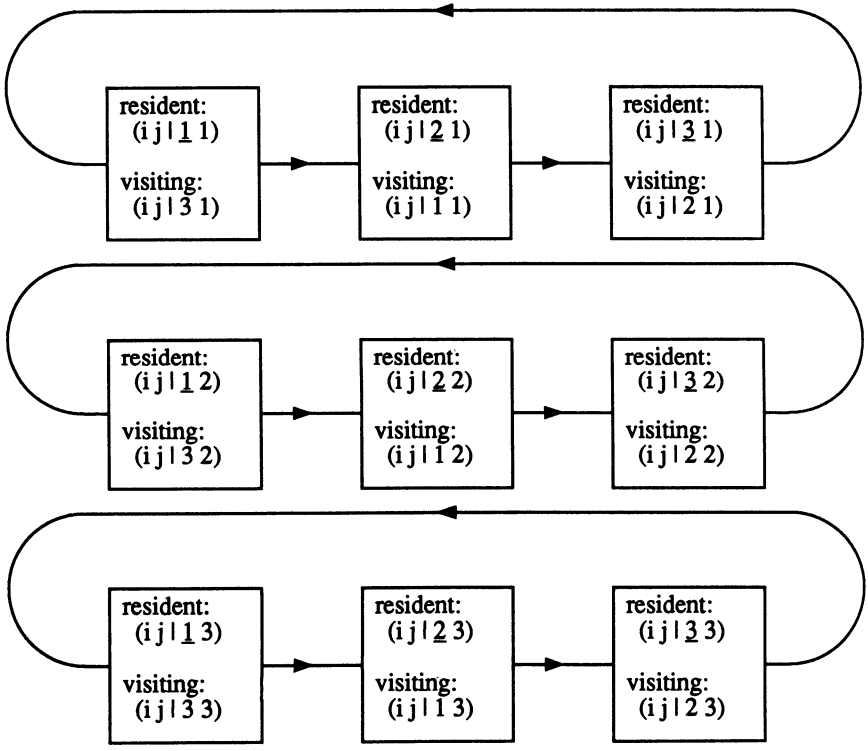
A final issue is the topology of interprocessor interconnection assumed by the parallel implementation. Although these programs are being implemented on a machine with hypercube connectivity, it is not necessarily prudent to assume the presence of that particular interconnection in this algorithm. If an efficient algorithm can be developed that requires only very simple topologies, such as one- or two-dimensional meshes, the algorithms will run on a wider range of parallel architectures.



Step 1. In-place transformation of first two indices.

Figure 12. Transformation for $n = 3$ on 3×3 grid of processors.

On the basis of the considerations listed above, the two-dimensional transformation scheme was chosen for implementation. This choice clarifies the reasons for the post-SCF integral distributions described in the integral evaluation section (Figures 4–7). The two indices that indicate the processor coordinates are those that require interprocessor communication to transform (analogous to σ in the one-dimensional case). The transformation procedure for the two post-SCF integral distribution schemes, $\lambda\sigma$ (Figures 4 and 6) and $\nu\sigma$ (Figures 5 and 7), are essentially identical, differing only in the order in which the indices are transformed. Hence, in the algorithm described here, only the $\lambda\sigma$ distribution will be considered. Further, in the following description the processing grid is assumed to be a square torus of processors ($m \times m$), where the number of basis functions, n, equals the number of nodes along each dimension (i.e., $m = n$). These constraints are not inherent in the algorithm, which will work efficiently for



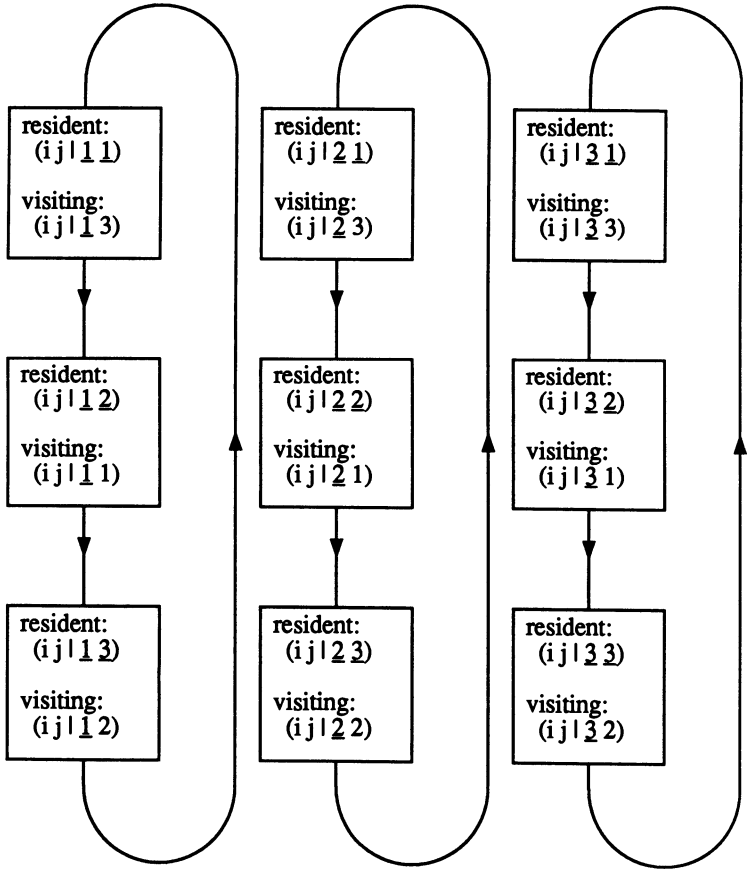
Step 2. Transformation of third index. Underlined numbers are MO indices.

Figure 13. Transformation for $n = 3$ on 3×3 grid of processors.

any number of basis functions provided ($n \geqslant m$) and on any torus supported by the interprocessor connectivity. (In general a hypercube architecture has as a subtopology a $2^i \times 2^j$ rectangular torus where $i+j \leqslant$ the total hypercube dimensionality.) The extension to cases with rectangular processing grids and cases where $n > m$ is straightforward.

At the outset of the integral transformation the integrals are distributed onto the processing nodes on the basis of their final two indices, $\lambda\sigma$, as shown in Figure 4. The transformation matrix C is the final converged SCF vector, which is broadcast from the host to all of the nodes using a broadcast tree.

The first step is the transformation of the first two indices, μ and ν



Step 3. Transformation of fourth index. Underlined numbers are MO indices.

Figure 14. Transformation for $n = 3$ on 3×3 grid of processors.

(Figure 12). This step is carried out as two quarter transformations. If each processor is assumed to hold only one $\lambda\sigma$ block and no integral symmetry is considered, then each quarter transformation requires n^3 multiplications. This yields a total of $2n^3$ for this entire step. However, if the $\mu\nu$ symmetry is exploited then the number of multiplications is reduced to $\frac{3}{2}n^3$.

The final two quarter transformations are more complicated since they will require interprocessor communication. Analogous to the transformation over σ in the one-dimensional case, these last two steps will require communication first across the processor rows (Figure 13) and then down the processor columns (Figure 14). Although these two steps are in principle straightforward, a number of details must be carefully considered in order to use memory efficiently and to avoid communication bottlenecks.

There are two general methods for carrying out the communication steps. The processor could either pass around MO integrals (with initial values of zero) that accumulate contributions from each of the other processors as they circumnavigate the row, or they could pass around raw (half transformed) AO integrals and accumulate MO integrals locally from each of the AO integral subsets as they pass by. (The latter strategy was described for the one-dimensional example.)

The chief distinction between these two strategies is the type of interprocessor communication allowed by each. In the first scheme all communication must be synchronous. That is, communication is not carried out while the nodes are involved in other computation. The reason for this is that each of the MO integral buffers must remain on each node until all contributions from the resident integral set have been added in. Only when all of these computations are completed is the integral buffer ready to be passed on to the next processor.

In contrast, when the raw AO integral sets are passed, the processor needs only to make a copy of the integral set before passing the integrals on to the next processor. This second scheme is obviously more efficient since it allows the overlap of communication and computation. However, the asynchronous scheme has some potential pitfalls. Since the integral buffers are passed on by each processor immediately, slow processors could potentially queue up many unreceived messages which can cause erratic behavior on the Hypercube. Nevertheless, in order to study the performance enhancement gained by asynchronous communication the second scheme was implemented in the final version of the integral transformation.

The code given in Figure 15 summarizes the operations performed during the third quarter transformation. The routine is passed the resident block of half-transformed integrals (*xints*), the SCF vector (*C*), the number of basis functions (*nbasis*), and the value of the third integral index for the resident AO (*mylambda*) and MO (k) integral blocks. Note that for the example in Figures 12–14, $mylambda = k$.

In order to allow asynchronous communication, the scratch array *buf* is divided into two segments, each able to hold a full integral block. Two variables, *outpoint* and *inpoint*, hold pointers to the two halfs of *buf*. The half indicated by *outpoint* holds the integral block about to be processed, and the half indicated by *inpoint* receives the incoming integral block.

The subroutine begins by initializing the pointers and copying its integrals from the array *xints* into the half of *buf* pointed to by *outpoint*,

```
subroutine trans(xints,buf,c,mylambda,k,nbasis)
comment: Perform the transformation on the index λ, xints
      holds the integrals, buf is a big scratch array, c is the MO coefficient matrix, mylambda is index of the
      resident integral block, k is the index of the MO block to be formed on the processor.
dimension buf(2,nbasis,nbasis), xints(nbasis,nbasis), c(nbasis,nbasis)
comment: initialize the pointers into buf .
inpoint=2
outpoint=1
comment: Copy the resident integrals into the first half of buf and zero
      xints.
do μ = 1,nbasis
      do ν = 1,nbasis
            buf(outpoint,μ,ν)=xints(μ,ν)
            xints(μ,ν)=0.0
      end do(ν)
end do(μ)
comment: lambda holds the λ value for the current integral block. Initially set to resident λ.
lambda = mylambda
comment: loop over nu mber of processors in row (= nbasis). With each pass process the contents of the half
of buf pointed to by outpoint.
do iloop = 1,nbasis
      comment: send the integral block about to be proceed to the neighboring
            processor and initiate receipt of a new integral block.
      call send(buf(outpoint))
      call recv(buf(inpoint))
      comment: sum into xints the contributions from outpoint half of buf.
      do μ = 1,nbasis
            do ν = 1,nbasis
                  xints(μ,ν) = xints(μ,ν) + C(k,lambda)*buf(outpoint,μ,ν)
            end do (ν)
      end do (μ)
      comment: update value of lambda to match incoming buffer.
      lambda = lambda - 1
      if (lambda.eq.0) lambda = nbasis
      comment: Check to make sure new integral buffer has been received.
      call waitchan
      comment: now swap values of outpoint and inpoint.
      call swap(outpoint,inpoint)
      end do(iloop)
return
end
```

Figure 15. FORTRAN-like code for the transformations of the third index.

zeroing *xints* as it does. The MO integrals will accumulate in *xints*. The routine then enters the main loop over the number of processors in each row, which equals *nbasis* for this example. Then first step is to send off a copy of the integral block that is about to be processed [stored in *buf*(*outpoint*)]. Next, *recv* is called to initiate the receipt of a new block of integrals which will be stored in *buf*(*inpoint*). After the communication steps are initiated, the contributions from the integrals in *buf*(*outpoint*) are summed into *xints*. Next the variable *lambda* is decremented in anticipation of the new integral block. Since it is possible that this communication is not yet complete, the routine calls *waitchan*, which blocks execution until the integral block is completely received. At this point the cycle is ready to begin again except that the roles of the two halfs of *buf* are switched—the newly received block of integrals will be sent on and processed, and the previously processed block will be overwritten by the incoming integral block. This switch can be accomplished by simply swapping the values of the pointers *inpoint* and *outpoint*.

After *iloop* has looped over all of the processors in the row, each processor has received all of the necessary contributions for the transformation of the third index. The only step remaining is the transformation of the final index σ. This step can be carried out in a way exactly analogous to the transformation of the third index, except that the interprocessor communication is down the processor columns rather than across the processor rows.

For clarity of presentation several important details have been neglected in this description. For instance, the code as presented requires each half of the buffer space *buf* to be able to hold a full integral block. In fact, the transformation can be carried out with a smaller buffer; it simply requires more passes through a code similar to this, each pass completely transforming only a subset of the integral block.

Each of the last two quarter transformations requires $n^3/2$ operations on each of the processing nodes. Since only n sequential communication steps are required for each quarter transformation, communication time will not dominate the execution time except for very small problem sizes. Each of the n^2 processors carries out $3(n^3/2)$ operations for the first half transformation over μ and ν and n^3 operations for the final half transformation over λ and σ, yielding a total computational complexity of $5(n^3/2)$ on each of the nodes for the entire four index transformation.

As mentioned earlier, for problems where the number of basis functions is not evenly divided by the number of nodes along each dimension of the processor mesh, the number of integral blocks assigned to each processor will be unequal and, hence, the computational load will be poorly balanced. An extreme example is shown in Figure 6, where four nodes must process four integral blocks each and one node is assigned only one

integral block. Since the lightly loaded processors will eventually have to wait for the more heavily loaded processors (during the third and fourth quarter transformations) the transformation of the $n=5$ case will require the same amount of time as the $n=6$ case. Of course for larger problem sizes where the number of basis functions is much larger than the number of nodes along each dimension of the grid, this load imbalance will contribute a proportionally very small contribution to the total running time. Nevertheless, it is worth investigating methods to improve the load balancing.

2.5. Moller–Plesset Perturbation Theory

The integrals produced by the transformation step have little intrinsic scientific value. They are useful only as input to subsequent computations. These are a large number of post-SCF procedures requiring the transformed integral list. The two most commonly used methods are configuration interaction and Moller–Plesset perturbation theory. For reasons that will be discussed shortly the perturbation theory calculations are easier to parallelize, so these were chosen as the first post-SCF methods to implement on the Hypercube. Considerations for the parallel configuration interaction algorithm will be discussed in a later section.

As described in the first section, Mollet–Plesset perturbation theory provides a series of additive corrections to the total electronic energy calculated by the SCF procedure. For all orders of the perturbation theory, these energy corrections have the same form: linear combinations of products of superintegrals formed from the transformed two-electron integrals and coefficients formed from the SCF orbital energies (the eigenvalues of the converged Fock matrix). In principle this step is easy to parallelize. Each processor asynchronously forms contributions to the energy correction from its resident list of transformed integrals. When completed, these partial contributions are sent back to the host processor, where they are summed together. The various details involved in this procedure will be covered in the following description of the second-order Moller–Plesset energy correction ($E^{(2)}$). Extension to higher orders of perturbation theory is straightforward.

The standard expression for $E^{(2)}$ is

$$E^{(2)} = \sum_{ab}^{\text{occ}} \sum_{rs}^{\text{vir}} \frac{|\langle ab| \, |rs\rangle|^2}{D_{abrs}}$$

This expression is meant to be used as a correction to unrestricted Hartree–Fock (UHF),[24] energies and hence, the summations are over spin orbitals. However, the SCF program implemented on the Hypercube

carries out restricted Hartree–Fock (RHF) calculations so that spin-independent spatial orbitals are produced. (The relative merits of UHF and RHF have been extensively debated.[47]) The conversion of the Moller–Plesset energy expressions from spin to spatial orbitals is quite easy. The resulting $E^{(2)}$ expression for closed shell systems is

$$E^{(2)} = \sum_{ab}^{occ} \sum_{rs}^{vir} \frac{(ar|bs)[2(ar|bs) - (as|br)]}{D_{abrs}}$$

where the integrals are the standard transformed two-electron integrals and D_{abrs} is formed from the final SCF orbital energies:

$$D_{abrs} = \varepsilon_a + \varepsilon_b - \varepsilon_r - \varepsilon_s$$

Hence, in order to form contributions to $E^{(2)}$ the nodes need access to the transformed integrals and the orbital energies. Since the number of orbital energies is the same as the number of basis functions (<150), it makes sense to send a separate copy to each of the nodes.

The transformed integrals are already distributed onto the nodes by the transformation program; however, the formation of each term in $E^{(2)}$ requires the presence on a single processor of a pair of integrals (corresponding to the pair of integrals required to form a single superintegral). These two integrals are related by the permutation of their second and fourth indices. An important question is whether these pairs of integrals will be available on the same processor or interprocessor communication will be necessary to collect them. Of course the answer depends on the final distribution of the transformed integrals.

As described in the previous sections two possible post-SCF integral distribution schemes have been investigated. The first distributes the integrals on the basis of their third and fourth indices (Figures 4 and 6), and the other distributes them on the basis of their second and fourth indices (Figures 5 and 7). For the computational steps through the integral transformation the chief difference between these two is that the latter requires an additional factor of 2 redundancy in the computation, storage, and transformation of the integrals. However, for the computation of $E^{(2)}$ these two distributions will lead to very different algorithms.

For the first of these schemes, the appropriate pairs of integrals will not be together on the same processor. For example, in Figure 4 the transformed integral $(33|11)$ is assigned to the node in the upper left corner, while its partner $(31|13)$ is assigned to the node in the lower left corner. This means that if the calculation of $E^{(2)}$ is to be carried out, some interprocessor communication will have to occur. The remaining question is whether this communication can be carried out in an orderly, efficient way.

```
subroutine mp2(numk,numl,l_list,k_list,nblocks)

    comment: This routine calculates the second order Moller Plesset energy from
        transformed integrals distibuted on the basis of their third and fourth indices.
        numl and numk contain the number of k and l index values for the integral
        blocks located on the processor. k_list and l_list are arrays containing the lists
        of k and l index values. nblocks contains the total number of valid integral
        blocks in the column of processors.

    dimension k_list(50), l_list(50)

    comment: Loop over resident integral blocks, sending valid blocks down the
        processor column.

    do 200 lcount = 1,numl
        l=l_list(lcount)
        do 100 kcount = 1,numk
            k=k_list(kcount)

            comment: Skip this block if l is not an occupied orbital or k is not an
                unoccupied orbital.

            if (l.gt.nocc) goto 200
            if (k.le.nocc) goto 100

            comment: Send valid blocks to lower neighbor.

            call sendblk(k,l)

            comment: Receive a block from upper neighbor

        50  call recvblk(owner,visiting_k,visiting_l)
            nblocks=nblocks-1

            comment: If received block is a resident block, make
                contributions, but don't pass it on.

            if (owner.eq.mynode) then
                call contrb(visiting_k,visiting_l,value)

                comment: If all blocks in column have been received, then quit.

                if (nblocks.eq.0) goto 500

                comment: Otherwise send out another resident block.

                goto 100

            end if
```

Figure 16. FORTRAN-like driver routine for type 1 second-order Moller–Plesset energy calculation.

```
            comment: Send visiting block to lower neighbor.
            call sendblkon
            comment: Calculate E(2) contribution from visiting integral block.
            call contrb(visiting_k,visiting_l,value)
            comment: Receive next block.
            goto 50
        100 continue
        comment: Now all valid resident integral blocks have been
            processed, but it is still necessary to wait for
            contributions from other processors' blocks.
        200 continue
        call recvblk(owner,visiting_k,visiting_l)
        call sendblkon
        call contrb(visiting_k,visiting_l,value)
        comment: Quit if last block has been processed.
        nblocks=nblocks-1
        if (nblocks.eq.0) goto 500
        goto 200
        comment: All that remains is to sum up the partial E(2) values on the host.
    500 call hostsum(value)
        return
        end
```

Figure 16. (*Continued*)

In this integral distribution scheme a given node has MO integrals with all possible values for the i and j indices and with values for k and l indices determined by the processor's position in the grid. Since the integral pairs are related by the permutation of the second and fourth indices, an integral block on a particular processor will have to access all blocks with the same k index, but having all possible values for the l index. Owing to the fact that the l index is used to assign the integral blocks to processor rows, the necessary integral blocks can be found on the other processors in the same column. Hence, the communication is very similar to that required in the final quarter transformation; buffers must be passed around the columns of the processor grid.

To clarify this somewhat complicated explanation, consider again the 3×3 example used in the transformation section. After the transformation, the integrals are distributed onto the processors as shown in Figure 4. The

node in the upper left holds the integral block $(ij|11)$. In order to form the corresponding block of superintegrals, this block will have to be combined with all integrals of the form $(i1|1j)$ with $i, j = 1, 3$. All of these integrals can be found in the first column of processors. A remaining question is how to assign $E^{(2)}$ contributions to the processors or, more generally, how to distribute the superintegrals. Since the superintegrals are defined as

$$\langle ab| \, |cd\rangle = (ac|bd) - (ad|bc)$$

a natural choice is to let the processor holding the first integral form and process the corresponding superintegral.

Note that for the calculation of the second- and third-order Moller–Plesset energy corrections not all the superintegrals need to be formed. For example, $E^{(2)}$ only has contributions from superintegrals of the form $\langle ab| \, |rs\rangle$ with a, b occupied orbitals and r, s virtual orbitals. These constraints will limit the number of integral blocks that need be sent around the processors.

The "driver" subroutine for the calculation of $E^{(2)}$ is given in Figure 16. Each processor loops over its resident integral blocks sending valid blocks down the processor column. For the calculation of $E^{(2)}$ valid blocks are those with k an occupied orbital and l a virtual orbital. (If a processor has no more valid blocks to send, it must still receive the remaining blocks being broadcast by the other processors in order to finish the contributions to its resident superintegrals.) After sending an integral block down the grid, each node receives an integral block and immediately sends on a copy of this block by calling the subroutine *sendblkon*. Next, *contrb* is called, which generates all superintegrals that can be formed from the resident and visiting integral blocks. Once formed, the superintegrals are not stored, instead they are immediately combined with the appropriate D_{ijkl} elements (which are formed as needed) and summed into the energy term. When all the nodes are finished calculating the partial $E^{(2)}$ terms, *hostsum* is called to sum the terms down a broadcast tree onto the host.

As mentioned earlier, the alternative integral distribution leads to a very different algorithm. In this scheme the integrals are assigned to processors on the basis of their second and fourth indices. In the 3×3 example (Figure 5) the upper left processor holds the integral block $(i1|k1)$ and the necessary partner integrals $(i1|11)$. Hence, with this integral distribution, no communication will be necessary to form the superintegrals. The algorithm for calculating $E^{(2)}$ is very simple. The routine simply loops over valid superintegrals, summing energy contributions into a buffer. When the loop is complete, the partial terms are summed onto the host processor.

Each of the two integral distributions has merits and disadvantages.

The first allows efficient integral evaluation and transformation, but requires n sequential communication steps to form the superintegrals. In contrast, the second scheme allows the formation of the superintegrals without communication, but adds a factor of 2 to the number of integrals that must be calculated and transformed. Which of these two schemes is ultimately the best is dependent on the hardware being used. For some of the presently available computers, on which communication and processing speeds are comparable and on which memory is at a premium, the former scheme is best. However, for future parallel computers, on which processing speeds will be much faster than communication speeds and memory will not be a limitation, the latter scheme will be optimal. Benchmarks results for both schemes are given in the next section.

2.6. Configuration Interaction

Although Moller–Plesset perturbation theory (MPPT) and configuration interaction (CI) are both methods to correct for SCF's exclusion of instantaneous electron–electron correlation in the electronic wave function, the actual computational procedures involved are very different. As discussed in the previous section, the computation of MPPT energies involves summations over the transformed integrals, so the parallelization is fairly straightforward. In contrast, CI calculations involve the formation and diagonalization of the CI matrix $\langle \Phi | H | \Phi \rangle$. This procedure is computationally difficult for both serial and parallel implementations since this matrix is very large—of order at least 100,000 for most systems of chemical interest. Fortunately this matrix is sparse and only the first few roots are ever desired, so that efficient iterative diagonalization methods can be used.

A commonly used iterative diagonalization method is that of Davidson.[48] In this algorithm the true eigenvectors C_j are approximated by a set of k basis vectors b_i:

$$C_j = \sum_i^k \alpha_{ij} b_i$$

where k is much less than the order of the matrix. Each iteration involves the construction and diagonalization of a $k \times k$ matrix $\mathbf{P}$ to form a new set of coefficients α_{ij}. Additionally with each iteration a new basis vector b is added to the expansion. The construction of the matrix $\mathbf{P}$ requires the multiplication of each of the basis vectors by the matrix to be diagonalized. Since a new basis vector is added with each iteration, this large matrix multiply must be carried out each iteration.

While the Davidson procedure provides an efficient way to compute the first few roots of the CI matrix, as described above it does not alleviate

the need to store the entire CI matrix. This can be avoided, however, by using a method known as direct CI. In direct CI as each matrix element contribution (a product of a transformed integral and a coefficient) is formed it is immediately combined with the appropriate elements in the guess vector. In this way when all of the CI matrix elements have been constructed the matrix multiplication is complete.

When the transformed integral list is evenly distributed onto the node processors the direct CI method is well suited for parallelism. Each processor calculates the coefficients for its set of transformed integrals. As each of these partial matrix elements are produced, they are combined with the appropriate elements in the guess vector. When the nodes have processed all of their two-electron integrals, the product vectors are summed together on the host. The new guess vector is then generated and broadcast to the nodes.

Although the algorithm should be easy to implement, it has the drawback that each of the processors must hold a copy of the guess vector (which is the same length as the CI expansion) as well as its set of MO integrals. Hence, owing to the limited memory currently available on the Hypercube the CI program has not yet been implemented.

3. Benchmark Results

3.1. Introduction

No matter how impressive the theoretical efficiency of an algorithm, the true test is its actual performance when implemented on a real computer. This is particularly true for parallel algorithms where unexpected bottlenecks can drastically reduce the anticipated performance. In order to test the algorithms presented in the previous section, they were implemented on a 32-processor Hypercube located at Sandia National Laboratories. Before presenting the actual results, it would be worthwhile to describe briefly the theory of measuring the efficiency of parallel algorithms.

The principal measure of performance is the elapsed time necessary to solve a real problem of interest. However, in evaluating parallel programs the speedup gained by adding additional processors is often a more informative measure. The speedup of a parallel program run on k processors, S_k is defined by

$$S_k = \frac{t_{\text{ref}}}{t_k}$$

where t_k is the time necessary to complete the computation on a

k-processor parallel computer, and t_{ref} is some reference time for the computation. It is common to take t_{ref} to be the time necessary to perform the computation on a single processor of the type used in the multiprocessor. Therefore, the ideal case is $S_k = k$, indicating that the computation runs a factor of k times faster when running on k processors.

The first set of benchmark results are for the integral and SCF programs. Results are given for both standard and direct SCF calculations. Next, the results for complete integral, SCF, transformation, and second-order Moller–Plesset energy calculations are given. Finally, the results are presented of an extensive battery of benchmark calculations for the most computationally complex of the programs implemented, the two-electron integral transformation.

As mentioned earlier, the chief constraint on the problem sizes that can be handled by the Hypercube is the relatively limited memory on each processing node. In the standard configuration, each node has 512 kbytes of RAM. After the node operating system is loaded, 322 kbytes remain for the user programs and data. The node programs for the integral and SCF calculations require 81 kbytes, leaving 240 kbytes. Since an n basis function problem requires the storage of $\sim n^4/8$ integrals (each requiring 8 bytes of memory), the maximum size SCF calculation that can be carried out using all 32 nodes is

$$\sqrt[4]{32 \times 240000} \approx 50 \text{ basis functions}$$

(Of course for a direct SCF calculation there is essentially no limit on the problem size since no integrals are stored.) The full set of node programs (integral evaluation through Moller–Plesset energy) require 119 kbytes, leaving 203 kbytes for integral storage. The two integral distribution schemes investigated require the storage of $n^4/2$ and n^4 integrals, respectively, corresponding to maximum problem sizes of 35 and 30 on a full 32-node Hypercube.

This limitation can be alleviated either by adding more memory to each node or by adding more processing nodes (so that each has a smaller fraction of the total integral list). Both of these options are now available for the Hypercube. The total number of processing nodes can be increased to 128, and the memory per node can be upgraded to 4 Mbytes. All calculations carried out for this chapter were on the standard configuration Hypercube.

In order to determine the program speedups it is necessary to run the benchmark calculations on various sized subsets of the Hypercube processors. The Hypercube allows the user to specify easily how many nodes to use for a given calculation. A natural choice of subsets used for most of these benchmarks are the "subcubes" of lower dimensions having

2^n processors for $n = 0$–5. However, using fewer than the full 32 nodes decreases the total available memory, so that the size of the test cases will be dictated by the smallest processor array in the benchmark sequence.

In order to test the performance on both large and small cases two test problems were chosen. The first is 24 basis function C_2H_2 for which a full Moller–Plesset calculation can be carried out on as few as 8 processing nodes. The second test case is 13 basis function H_2O, which is sufficiently small to fit on a single processing node. The details of these test cases are given in Appendix B.

3.2. SCF Results

Since the host-to-node communication rate is about 1/3 that for node-to-node, for calculations involving no post-SCF steps it is more efficient to have one of the node processors take the role of the host, receiving and diagonalizing the Fock matrix. (Note that this would be complicated for post-SCF computations since the transformation requires a full rectangular grid.) This strategy has been employed for the SCF benchmarks, and hence the processor grids have size $2^n - 1$ ($n = 1$–5).

The integral and SCF benchmark results for the C_2H_2 test case are given in Table 5. Each part of this table has three columns of data. The first contains results for the two-electron integral evaluation. The second gives results for the SCF step, and the third lists results for the construction of the partial Fock matrix.

The first part of Table 5 gives the total timings in seconds. Obviously the timings in the second and third columns vary from node to node (unless load balancing is perfect). The timings given are those for the slowest processors. The second part of the table lists the speedup ratios derived from the timings in the first part of the table. The values should be compared with the numbers in the "ideal speedup" column. These data show that the integral evaluation and Fock matrix formulation steps have nearly perfect speedups. The slightly superlinear speedups in the table are due to changing load balances for different sized test cases.

Although the results for the Fock matrix formation step are very good, the speedup for the full SCF calculation is not very encouraging. The efficiency on 31 processors is less than 30%. Since the parallelized portions show good speedup, the problem must be that the serial portion (the communication and diagonalization of the Fock matrix) must be dominating the computation time. The communication benchmarks (Appendix A) indicate that the collection of the partial Fock matrices is rapid, so that the bottleneck must be the diagonalization. For this test case ($n = 24$) this is not surprising. As discussed in the SCF section, when the number the number of nodes (m) is comparable with the number of basis functions (n),

Table 5. SCF Benchmark Results for 24 Basis Function C_2H_2

Total execution time (s)			
No. of nodes	Integral	SCF	Fock matrix
3	811.94	1289.20	7.86
7	352.06	706.21	3.14
15	158.66	479.92	1.49
21	79.14	378.62	0.70

Speedups				
No. of nodes	Integral	SCF	Fock matrix	Ideal
3	1.00	1.00	1.00	1.00
7	2.34	1.83	2.25	2.33
15	4.92	2.69	5.05	5.00
31	10.0	3.40	10.31	10.33

Load imbalance (%)			
No. of nodes	Integral	SCF	Fock matrix
3	1.4	—	11.9
7	5.9	—	4.9
15	9.1	—	17.3
21	11.3	—	14.6

the parallel Fock matrix formation and the serial diagonalization steps have the same computational complexity. Although for much larger problem sizes ($n \gg m$) the SCF speedup should be good, it is clearly necessary to parallelize the diagonalization step in order to allow efficient use of large processor arrays.

The third part of Table 5 gives the percentage differences in the timings for the fastest and slowest processors indicating the quality of the load balancing. The results are quite good, especially considering the small size of the test case.

Table 6 gives the same results for the 13 basis function H_2O test case. The results are pretty much the same as those for C_2H_2 except that the smaller size of the problem leads to poorer speedups and load balancing on large numbers of processors. A surprising and rather curious result is the slight superlinear (i.e., better than ideal) speedups seen for the first few numbers of processors. Since these speedups have been calculated relative to single node timings this cannot be the result of poor load balancing in the reference calculation (as was the case for the C_2H_2 benchmark). At this

Table 6. SCF Benchmark Results for 13 Basis Function H_2O

Total execution time (s)			
No. of nodes	Integral	SCF	Fock matrix
1	364.83	5167.36	2.05
3	116.42	2200.32	0.67
7	50.93	141.74	0.34
15	25.97	113.94	0.26
31	11.78	101.58	0.16

Speedups				
No. of nodes	Integral	SCF	Fock matrix	Ideal
1	1.00	1.00	1.00	1.00
3	3.13	2.35	3.05	3.00
7	7.16	3.65	6.10	7.00
15	14.05	4.535	8.00	15.00
31	30.98	5.09	12.80	31.00

Load imbalance (%)			
No. of nodes	Integral	SCF	Fock matrix
1	—	—	—
3	4.5	—	7.1
7	15.0	—	23.8
15	28.2	—	56.3
21	16.8	—	72.5

time this anomalous result has not been definitely attributed to a specific cause. A careful study of the algorithm has found nothing that would preferentially degrade the single processor performance. This indicates that some of the problem arises from some feature of the node hardware. A likely candidate is associated with how the processor handles large data arrays. The size of the data sets on each node is inversely proportional to the number of nodes participating in the calculation. Hence, if references into very large data arrays are at all slower than references into small arrays, the algorithm's performance would be degraded when running on fewer processors. The fact that memory access speed is inversely proportional to memory size has been used as an argument for the possibility of superlinear speedups on parallel computers.[49] However, more extensive benchmarks of the Hypercube would be needed to verify that this is the phenomenon being observed.

3.3. Direct SCF

The benchmark results for parallel direct SCF calculations on C_2H_2 and H_2O are given in Tables 7 and 8. As before, these data are in three parts of the tables, giving total run times, speedups, and percentage load imbalance. Each of these parts has two data columns. The first gives results for the complete calculation, while the second contains results for a single iteration of the parallelized portion of the program. The parallelized portion involves both the integral evaluation and the formation of the partial Fock matrix. As before, the timings given are for the slowest processor. Note that the reference calculation for C_2H_2 was run on three processors (rather than one) because of time limitations; memory is not a limitation for direct SCF calculations.

As might be expected from the previous results, the speedups are very good. The parallelized portions show nearly perfect speedups over the entire range of node configurations. The speedup for the complete calculation is degraded on large numbers of processors (particularly for the smaller test case) as the serial diagonalization step grows to dominate the

Table 7. Direct SCF Benchmark Results for 24 Basis Function C_2H_2

	Total execution time (s)	
No. of nodes	Total SCF	Integral and Fock matrix
3	19623.50	838.99
7	8573.07	358.88
15	4232.83	170.42
31	2248.43	84.02

	Speedups		
No. of nodes	Total SCF	Integral and Fock matrix	Ideal
3	1.00	1.00	1.00
7	2.29	2.34	2.33
15	4.64	4.92	5.00
31	8.73	9.99	10.33

	Load imbalance (%)	
No. of nodes	Total SCF	Integral and Fock matrix
3	—	1.4
7	—	8.0
15	—	9.0
31	—	11.1

Table 8. Direct SCF Benchmark Results for 13 Basis Function H_2O

Total execution time (s)			
No. of nodes	Total SCF	Integral and Fock matrix	
1	11555.23	368.90	
3	3749.12	117.82	
7	1691.25	51.52	
15	909.34	26.27	
31	479.1	12.37	
Speedups			
No. of nodes	Total SCF	Integral and Fock matrix	Ideal
1	1.00	1.00	1.00
3	3.08	3.13	3.00
7	6.83	7.16	7.00
15	12.71	14.14	15.00
31	24.12	29.83	31.00
Load imbalance (%)			
No. of nodes	Total SCF	Integral and Fock matrix	
1	—	—	
3	—	2.9	
7	—	14.9	
15	—	28.1	
21	—	19.9	

computation time. This problem can only be alleviated by parallelizing the diagonalization step. Once again the results show slight ($<5\%$) superlinear speedups for small numbers of processors.

3.4. Post-SCF Benchmarks

This section describes benchmark results for the full Moller–Plesset energy calculations. This procedure involves four steps: integral evaluation, SCF, integral transformation, and the $E^{(2)}$ calculation. Benchmark calculations were run for both the C_2H_2 and H_2O test cases using both of the proposed Moller–Plesset algorithms. These results are given in Tables 9–12 in the same format as the results in the previous two sections.

The integral and SCF results are very similar to those presented in the previous sections. As anticipated, the speedups and load balancing for these steps are worse, owing to the more complicated integral distribution con-

Table 9. Type 1 Moller–Plesset Benchmark Results for 24 Basis Function C_2H_2

Total execution time (s)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
8	1213.02	863.42	9.28	341.84	4.59
16	649.12	804.93	4.85	163.70	2.66
32	399.15	730.43	2.58	81.26	2.19

Speedups

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT	Ideal
8	1.00	1.00	1.00	1.00	1.00	1.00
16	1.87	1.07	1.91	2.09	1.73	2.00
32	3.04	1.18	3.60	4.21	2.09	4.00

Load imbalance (%)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
8	14.8	20.8	15.2	—	—
16	27.3	18.2	21.8	—	—
32	47.6	25.9	34.8	—	—

Table 10. Type 2 Moller–Plesset Benchmark Results for 24 Basis Function C_2H_2

Total execution time (s)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
16	1242.32	945.88	5.09	385.60	0.48
32	764.34	925.42	3.09	190.58	0.24

Speedups

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT	Ideal
16	1.00	1.00	1.00	1.00	1.00	1.00
32	1.63	1.02	1.65	2.023	2.00	2.00

Load imbalance (%)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
16	24.4	35.8	43.4	—	—
32	47.2	34.9	63.7	—	—

Table 11. Type 1 Moller–Plesset Benchmark Results for 13 Basis Function H_2O

Total execution time (s)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
2	698.03	325.66	3.55	76.96	3.76
4	341.22	266.11	1.78	38.69	1.92
8	190.99	257.78	0.96	21.41	1.87
16	109.66	227.94	0.46	12.18	1.14
32	60.00	209.66	0.32	6.26	0.88

Speedups

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT	Ideal
2	1.00	1.00	1.00	1.00	1.00	100
4	2.05	1.22	2.00	1.94	1.96	2.00
8	3.66	1.26	3.70	3.60	2.01	4.00
16	6.37	1.43	6.34	6.23	3.31	8.00
32	11.63	1.55	11.10	12.30	4.27	16.00

Load imbalance (%)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
2	9.6	20.6	12.6	—	—
4	18.3	23.5	21.6	—	—
8	39.0	28.9	33.3	—	—
16	53.8	25.9	45.7	—	—
32	61.3	18.0	65.0	—	—

straints imposed by the transformation step. The particularly poor speedup for the SCF step is due to the fact that a more lengthy matrix diagonalization procedure is required since the transformation requires the full set of eigenvectors.

The speedup results for the integral transformation step are quite good. For the larger C_2H_2 test case the speedups are ideal (or even slightly superlinear) for the entire range of processor configurations. The reason for the superlinear results is well understood. It is a consequence of the fact that interprocessor messages are limited to 16 kbytes on the Hypercube. When small numbers of processors are involved in the transformation of a large test case, many integral blocks must be assigned to each node. For sufficiently large cases, the communication of the partially transformed integrals in the final two steps of the transformation cannot be done in one message passing step. Instead, the integrals must be passed in several communication steps requiring extra overhead for the packing and sending of

Table 12. Type 2 Moller–Plesset Benchmark Results for 13 Basis Function H_2O

Total execution time (s)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
2	1292.69	372.26	3.30	178.38	0.45
4	631.36	317.33	1.66	87.04	0.24
8	353.66	299.22	0.99	48.27	0.16
16	202.94	274.90	0.59	27.25	0.11
32	111.01	237.38	0.40	13.98	0.06

Speedups

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT	Ideal
2	1.00	1.00	1.00	1.00	1.00	1.00
4	2.05	1.17	1.98	2.05	1.87	2.00
8	3.66	1.24	3.32	3.70	2.80	4.00
16	6.37	1.35	5.72	6.54	4.00	8.00
32	11.65	1.57	8.24	12.76	7.00	16.00

Load imbalance (%)

No. of nodes	Integral	SCF	Fock matrix	Transformation	MPPT
2	9.6	33.5	22.3	—	—
4	9.4	36.6	16.3	—	—
8	39.1	39.9	58.1	—	—
16	51.5	38.0	69.4	—	—
32	62.7	29.3	88.0	—	—

the message packets. Since this multistep communication is necessary only when a small number of processors are being used, these cases will have degraded performance. This suggests that the parallel transformation program does not yield good few-processor reference timings. Instead, the speedups should be calculated relative to a transformation program better suited for single-processor operation. This strategy was used for the more extensive transformation benchmarks given in the next section.

As described in the previous section two different Moller–Plesset algorithms were investigated. One (type 1) involved interprocessor communication in the formation of the superintegrals. The other (type 2) required no communication at the expense of an extra twofold redundancy in the number of integrals evaluated and transformed. The type 1 results are given in Tables 9 and 11 and the type 2 results in Tables 10 and 12. These results show that the type 2 algorithm is 5–10 times faster than the type 1. Of course, the second-order Moller–Plesset calculation requires so little time

that the extra time spent in the integral evaluation and transformation steps far outweighs any gain by using the type 2 algorithm. However, for higher orders of perturbation theory, which have greater computational complexities than the transformation, the type 2 algorithm should be overall more efficient.

Note that the relatively poor speedups exhibited by the Moller–Plesset program are the result of the limited number of occupied orbitals in the two test cases. Since the $E^{(2)}$ expression involves only integrals of the form $(ir|js)$ with i, j occupied and r, s unoccupied, the speedup is largely limited by the number of occupied orbitals in the test case.

Load-balancing data are not listed for the transformation and Moller–Plesset steps since these procedures involve explicit synchronization of the processing nodes. Thus, overall load balancing would be difficult to measure.

Table 13. Summary of Execution Times in Seconds for Parallel Transformation Program

Processors		Basis functions						
	Mesh	8	16	20	23	24	25	32
1	2×2	11.97	—	—	—	—	—	—
2	1×2	6.06	175.65	—	—	—	—	—
2	2×1	6.05	175.70	—	—	—	—	—
4	1×4	3.20	85.18	267.57	—	—	—	—
4	2×2	3.14	84.51	265.76	—	—	—	—
4	1×4	3.20	85.22	267.63	—	—	—	—
8	1×8	1.82	43.04	159.09	277.23	321.44	—	—
8	2×4	1.71	42.19	128.94	284.54	315.87	—	—
8	4×2	1.74	42.14	128.88	277.06	315.89	—	—
8	1×8	1.92	43.14	159.26	272.06	321.49	—	—
16	1×16	—	23.68	107.46	185.41	216.53	260.46	—
16	2×8	1.17	21.81	77.47	140.24	155.65	262.63	—
16	4×4	1.12	21.44	64.00	136.00	154.10	246.13	—
16	8×2	1.18	21.66	77.54	137.38	155.65	254.14	—
16	16×1	—	23.25	107.65	182.21	216.98	260.27	—
32	1×32	—	—	—	—	—	—	352.54
32	2×16	—	12.35	53.01	95.14	105.81	132.75	327.73
32	4×8	0.90	11.34	38.80	68.61	77.65	139.36	320.56
32	8×4	0.86	12.08	38.80	68.61	77.62	139.25	320.54
32	16×2	—	12.34	53.17	95.50	105.84	132.74	327.79
32	32×1	—	—	—	—	—	—	352.32

3.5. Transformation Benchmarks

The reason for developing these parallel quantum chemistry programs was to be able ultimately to study very large chemical problems on the massively parallel computers soon to be available. For such problems the computational time will truly be dominated by the most computationally complex steps. Hence it is important to benchmark carefully those steps that will become future bottlenecks. For this reason an extensive series of benchmark calculations was carried out on the two-electron integral transformation. In order to save the time and memory space required by the integral and SCF calculations, the transformation was carried out on a set of "dummy" integrals, which were a simple function of the AO indices. The transformation matrix C was simply a matrix with all entries equal to 0.5 to simplify the checking of the results. Table 13 summarizes the timings for various meshes and numbers of basis functions.

In order to determine accurate speedups, a serial transformation program was written. Since this program required too much memory to run large test cases on a single Hypercube node, it was implemented on an IBM 3081 K at the University of California, Berkeley. A series of benchmark calculations revealed that the mainframe executed the transformation

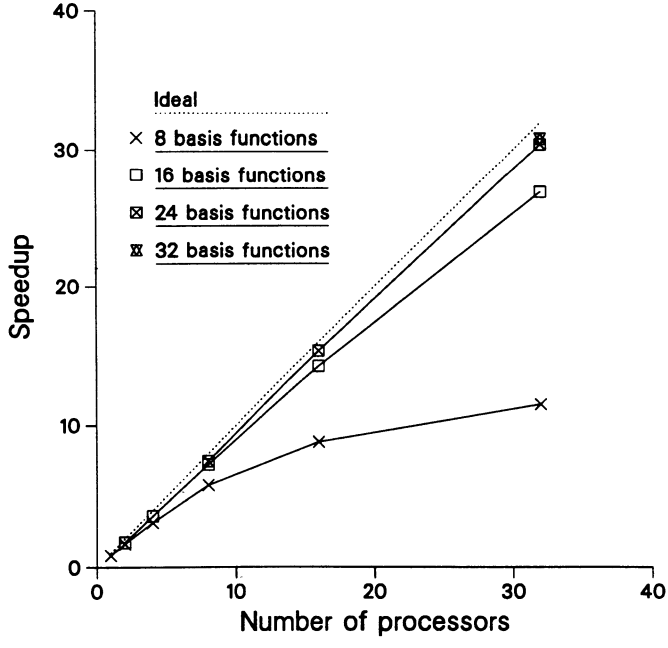


Figure 17. Speedup curves for test cases with even load balancing.

103.5 times faster than the Hypercube node. Thus, the speedup on k processors is defined as

$$S_k = \frac{t_{\text{ref}}}{t_k} \times 103.5$$

where t_k is the computer time on k nodes and t_{ref} is the time for the serial calculation on the IBM.

The speedups calculated in this way are given in Figures 17 and 18. Figure 17 gives speedup curves for those basis sets sizes where the load evenly divides onto the processor grid. The results for $n = 24$ and $n = 32$ are nearly ideal. The curve for $n = 16$ shows some falloff, but the efficiency is still greater than 90% on all 32 processors. For the smallest case, $n = 8$, there is significant falloff, indicating that the problem is so small that the communication is becoming a bottleneck.

Figure 18 shows speedup curves for several problem sizes where even load distribution is not possible for some of the meshes. Since the overall speed of the transformation is limited by the most heavily loaded processor, these results are not as good as those for evenly loaded problem sizes. As $n = 24$ is an evenly balanced case, $n = 23$ and $n = 25$ are two extremes in poor load balancing. For $n = 23$ a few processors have less

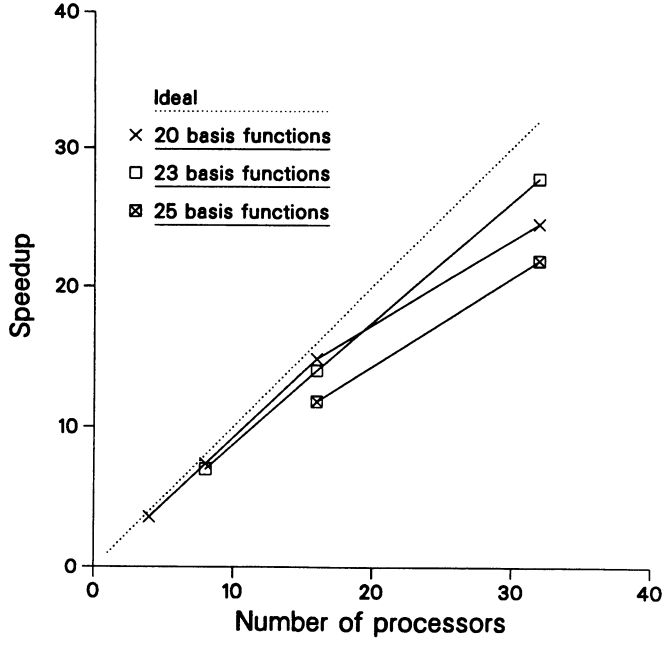


Figure 18. Speedup curves for problem sizes with unequal load distributions.

work than the majority, so the speedup is not seriously degraded. However, for $n = 25$ only a few processors have extra work, but the remaining majority must wait for these to catch up, leading to a more severe performance degradation. Nevertheless, these results are still quite good considering the relatively small size of the test cases.

3.6. Summary

Despite the constraints on problem size due to the limited memory on each node, good performance speedups were found for the integral evaluation and transformation steps as well as the parallel portions of the SCF. The SCF benchmarks clearly indicate the need to parallelize the diagonalization of the Fock matrix. Although the test cases were too small to yield reliable benchmarks for the Moller–Plesset step, the speedup on smaller processor meshes is encouraging.

4. Conclusions and Future Directions

The work described in this chapter represents a first step in the development of a general system of quantum chemistry programs able to exploit the capabilities of massively parallel computers. Just as the development of efficient serial algorithms for quantum chemistry has taken many years, the remaining steps will require the efforts of many researchers over a number of years. Much of the remaining work has been alluded to in the preceding sections. The following paragraphs will describe some of the continuing work currently being pursued by the authors.

As described in the previous section, most of the programs implemented should give good performance on the parallel computers available in the foreseeable future. A notable exception is the SCF step, where the diagonalization step can become a bottleneck on large processor grids. In order to rectify this situation work has begun in implementing the parallel diagonalization algorithm described in this chapter.

A second direction of current work is to complete the implementation of the third- and fourth-order Moller–Plesset energy corrections. Once implemented, these methods will allow truly "state-of-the-art" calculations to be carried out in parallel.

Although not currently under development, the next logical step will be the implementation of a general configuration interaction program. This method has a number of advantages over Moller–Plesset perturbation theory, probably the most important being that it yields a variational upper bound on the total energy. Additionally, the general methodology of

forming and diagonalizing a large Hamiltonian matrix is applicable to a large number of problems in the physical sciences.

Finally, a straightforward but practically useful task will be the implementation of programs to compute the first and second derivatives (with respect to nuclear coordinates) of these various total energies (SCF, MPPT, and CI). These procedures are well worked out for serial computers, and the bottleneck steps involve the computation of large integral lists and the solution of large systems of linear equations, both of which should easily parallelize.

Although the algorithms described in this chapter have been designed to work efficiently on distributed memory parallel computers, they are sufficiently general that they should provide good performance on a wide range of parallel architectures. Work is currently under way to get these programs running on two vastly different parallel computers.

The first of these is a ten-processor Elxsi 6400 located at Sandia National Laboratories. Although this computer has a shared memory architecture and hence no explicit interprocessor communication channels, communication can be mimicked by routines using the shared memory. Work is under way writing Hypercube compatible communication routines. These routines will require computational overhead to mimic effectively communication channels, but this should not adversely affect the performance of the algorithms since they were specifically designed for systems with relatively slow interprocessor communication. This suggests that all parallel algorithms should be targeted for distributed memory computers since shared memory computers can effectively mimic distributed memory architectures, while the converse is not true.

A second type of parallel computer onto which these programs are being implemented is not a tightly coupled set of homogeneous processors as are the Hypercube or Elxsi. Instead, this "computer" is a large group of workstations linked together by a local area network. Such collections of fairly powerful workstations (each comparable to a DEC VAX 11/780) linked by communication channels to each other and shared resources (such as printers and disk drives) are becoming increasingly popular and will soon be a ubiquitous presence in businesses and research institutions. Although these workstations are individually too small to perform large-scale scientific computations, together they represent an enormous computing resource (especially since they are often idle during evenings and weekends). For example, the network of $\sim$200 SUN 3/50 workstations currently in place at the University of California at Berkeley is, as a unit, several times the power of the Cray X-MP at the central computing facility.

All that is required to use these large networks as a single distributed memory computer is to write a series of Hypercube compatible communication routines using the existing network protocols. The relatively

slow network communication speeds should not seriously degrade the performance of the algorithms described here. A set of preliminary routines have been written by Curtis Janssen at Berkeley. The speedup curve for a 26 basis function integral and SCF calculation running on 1–8 SUN 3/50 workstations is given in Figure 19. Benchmark calculations will be carried out on much larger arrays of workstations as soon as a more efficient set of communication routines is complete.

In addition to the development of parallel algorithms for the standard quantum chemical methods, an important direction of study is the development of entirely new approaches to molecular quantum mechanics which involve inherently parallel techniques. One promising method that has been developed during the past decade is Quantum Monte Carlo (QMC).[50] The basic idea behind QMC is to recast the Schrödinger equation as a diffusion equation, which is then solved by a stochastic algorithm. This algorithm involves carrying out standard Monte Carlo updates on a large ensemble of possible configurations of the system (i.e., electron distributions). After a certain number of generations, configurations are either

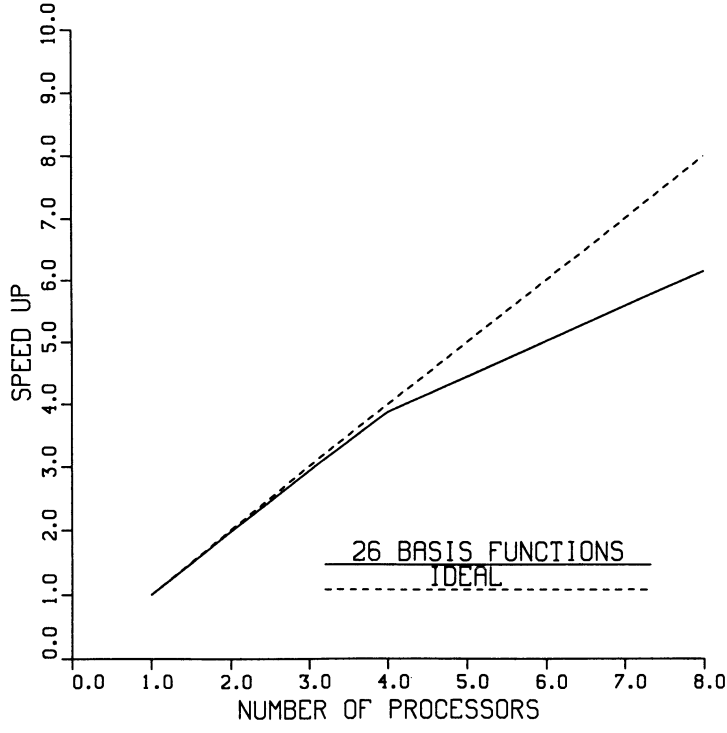


Figure 19. SCF speedup curves for a network of Sun Workstations.

discarded or replicated on the basis of a Boltzman weighting of their total energy. (This is called the branching step.) Then the Monte Carlo procedure is again carried out on the new ensemble of configurations. In this way, the ground state wave function eventually evolves.

Since this algorithm involves the concurrent simulation of a large ensemble of configurations it is ideal for parallelization. Each processor is assigned a configuration or set of configurations for the Monte Carlo updates. Interprocessor communication is required only during the branching step. Unfortunately, there are still some problems using QMC on fermion systems related to the selection and convergence of the wave function's nodal structure. However, these problems are being addressed so that QMC should be a very promising method for parallel implementation in the coming years.

Another potentially promising method involves the solution of the SCF equations using pseudospectral methods originally developed for hydrodynamic simulations.[51,52] The basic strategy is to construct different parts of the Hamiltonian matrix (in this case the Fock operator) in different representations. Certain parts of the Hamiltonian are best calculated in the spectral (i.e., basis set) representation. These terms include the kinetic energy terms involving derivatives with respect to position. Other terms including the electron–electron repulsion terms are best constructed in the spatial representation (i.e., on a three-dimensional grid). The collocation method is used to transform between these two representations.

The net effect of this approach is that the number of two-electron integrals is reduced from n^4 to n^3. This is potentially a big advantage for the development of parallel implementations since the amount of memory on each processor is one of the chief limitations on the size of problems that can be studied.

Of course a lot of research on parallel processing is being carried out by the computer science community. Of particular interest to computational physical scientists is the work on software to parallelize automatically serial programs. Ideally programs should not have to be specifically tailored to utilize efficiently a particular computer. Instead the compiler should be able to restructure the program for optimal performance regardless of whether the architecture is scalar, vector, or parallel.

Vectorizing compilers have been under development for nearly a decade and are now fairly adept at reordering loop structures to optimize vector performance. By comparison, automatic parallelization compilers are still quite new and limited in capability. Those currently available seem capable of carrying out efficiently very fine-grained parallelization. More specifically, these systems distribute the program on an instruction by instruction basis, for example, distributing the successive cycles though a do-loop. While this approach has proven successful so far, the fine-grained

nature of the parallelism requires very high interprocessor communication rates and hence will be limited to relatively small numbers of processors.

In order to achieve the very large performance enhancements promised by massively parallel computers, much coarser-grained parallelism will have to be used. However, the automatic detection and exploitation of coarse-grained parallelism is very difficult. The problem is that this requires knowledge of the control structure and data dependencies of the program, and such information is very hard to extract from programs written in standard scientific programming languages such as FORTRAN. Unfortunately this difficulty is not due to superficial features, but rather due to the fundamental structure of these languages.

A particularly problematic feature of languages like FORTRAN is that all variables point to memory locations ("pass by pointer"). While this has the advantage of saving memory, it means that the input values to functions and subroutines can (and often are) modified by the routine. Such modifications are called side effects and they greatly complicate the problem of parallelization. For example, consider the following segment of code:

$$D = \mathit{func\ 1}(A, B)$$

$$E = \mathit{func\ 2}(A, C)$$

Potentially *func 1* and *func 2* could be evaluated in parallel. However, since *func 1* may modify A, extensive analysis of *func 1* is necessary before parallelization can be implemented.

These difficulties are sufficiently serious that it is unlikely (at least in the near future) that a fully automated compiler will be available that is capable of restructuring efficiently a program for a distributed memory computer. Of course many software tools will be available to aid in the development of parallel programs, but the burden of determining how to distribute the program will still be on the programmer.

In order to overcome some of the difficulties associated with standard scientific programming languages, computer scientists have developed a new type of programming language known as functional languages.[53] In these languages the programmer works entirely with instructions that act like mathematical functions, taking particular inputs and returning outputs. An example of a simple function is $sum(A, B)$ which returns $A + B$. Note that A and B are unmodified (i.e., there are no side effects). From simple operations such as *sum* the programmer constructs more and more complex functions until ultimately the program itself is defined in terms of these high-level functions. For example, the following is a hypothetical SCF program:

$$scf\_energy = sum(elect\_energy(nuc\_coords, basis), nucl\_energy(nuc\_coords))$$

where the total SCF energy is calculated as the sum of two functions calculating the electronic and nuclear repulsion terms.

Such an approach has a number of useful properties. It produces well-structured, easy to read programs and it easily allows the construction of large programs from sets of smaller programs. Further, functional programs are in principle easily parallelized. The reason for this is that since the functions have no side effects, all terms in a given expression can be evaluated concurrently. In the example given above, the functions *elect_energy* and *nucl_energy* can be evaluated simultaneously.

Despite these advantages, functional languages are somewhat difficult to use and are inefficient on standard computer architectures. Hence, they are still not universally advocated. Exactly how much of the task of parallelization will eventually belong to the programmer is still unclear; however, it is likely that efficiently programming parallel computers may always be more difficult than serial programming. Nevertheless, if large performance increases are needed, this extra effort is justified.

A final question that should be addressed in this chapter is whether there are ultimate limits on the speed and accuracy of scientific computation. This question is more subtle than that of the limitations on processing speeds. Parallel computers can in principle be made arbitrarily fast by linking together arbitrarily large numbers of processors. Hence, the real limit is not the maximum potential speed of the parallel computers, but instead the ultimate degree to which scientific problems can be parallelized. Of course this largely depends on the scientific problem at hand. Therefore, for simplicity consider the numerical computation of the dynamics of a many-body system. Since this computation is simply a direct simulation of nature, the limits on its parallelizability are tied to the deeper, more general question: to what extent are natural processes parallelizable? That is, to what extent can nature be decoupled into separate processes interacting only locally?

The difficulty with this decoupling is that at the scales at which nature is usually simulated, particles are assumed to exert instantaneous action at a distance. Thus, for a system spatially mapped onto a grid of processors the motion of particles on distant processors will to some degree be coupled. This coupling will require interprocessor communication, putting tight constraints on the number of processors that can be efficiently used.

All natural forces attenuate with distance so that there is a cutoff distance beyond which particles can be considered noninteracting. Therefore the limit on the numbers of processors that can be efficiently used is determined by the ratio of the distance of significant interactions to the size of the system. For example, if the system being simulated is a relatively small cluster of very massive particles, all particle motions will be tightly coupled so that the system will not be amenable to massive parallelism. In contrast,

systems with no action at a distance such as continuum fluids and "billiard-ball" models are ideal for massive parallelism. For such systems the only limitation will be the communication required to calculate global system properties such as energy, average momentum, and so on.

Since interprocessor communication limits the performance of parallel computers, one strategy would be to shorten the message size and data routes. This is the motivation for radical new computer architectures which involve thousands of tiny few-bit processors. Since the processors are very small many can be fit onto a single integrated circuit so that the communication channels are very short. Moreover, the processors are designed such that the largest useful messages are only a few bits.

In order to use such machines efficiently, a very nontraditional approach to scientific simulation is required.[54] Rather than stylize large-scale dynamics into complex partial differential equations to be solved by numerical schemes, the microscopic dynamics are encoded into very simple (few-bit) rules describing the local interactions of particles or small spatial regions. The large-scale behavior of the system is determined by studying very large aggregates of the interacting subregions. Since the interaction rules are simple and local, such algorithms are easily mapped onto these novel computer architectures.

The first major application of this strategy has occurred only in the past year when Pomeau and co-workers[55] discovered that the large-scale dynamics of two-dimensional incompressible fluid flow can be reasonably modeled by a large ensemble of simply interacting point particles moving on a hexagonal mesh. This so-called hexagonal lattice gas model is straightforward to parallelize and has already been efficiently implemented on a 65,536 processor Connection Machine.[56] Despite its recent development, this model has stirred a great deal of scientific research and raised many fundamental questions about hydrodynamics and computational simulations.[57] Work is currently underway to extend this approach to more complex systems including multicomponent fluids[58] and plasmas.[59]

Despite this success, it is still not clear whether these methods will be applicable to strongly interacting, noncontinuum systems. Of course the ultimate simulation of nature would be carried out at such a small scale that instantaneous action at a distance would not occur—particles would interact directly by exchange of virtual particles. However, at these scales the particles themselves would be nonlocal wavepackets, leading to even deeper questions about the simulation of fundamental interactions. Hence, the question of whether there exist ultimate limits on the scale and accuracy of scientific simulations is not yet answerable; however, it is clear that the attempt to reach this limit will uncover clues to the ultimate goal of science, an understanding of the fundamental workings of nature.

Appendix A: Hypercube Benchmarks

A series of benchmark calculations were carried out to test the processing speed and interprocessor communication rate of the Intel Hypercube. Table A.1 gives the timings for various operations on double precision real variables. These benchmarks were run on the processing nodes using the system *clock* function to time the operations. The timing values are in milliseconds and indicate the amount of time to carry 100,000 repetitions of the operation. In this table R indicates a double precision real variable, and the final table entry is the overhead time for a do-loop of length 100,000. With do-loop overhead taken into account the processing speed of each node is 19,230 double precision multiplications per second. Hence if all 32 nodes were used with 100% efficiency, the overall processing speed of the Hypercube would be approximately 0.62 MFLOPS (million floating point operations per second) or about the speed of 6 DEC VAX 11/780's.

Determining the interprocessor communication rates is a more complicated procedure. Since the clocks on the different nodes are not necessarily synchronized, it is not possible to time directly the communication of messages between nodes. Instead, it is necessary to time the cycle of a message from a node to its neighbor and then back to the original node again. Since this round trip involves two complete message passing steps, the one-way communication rate can be calculated.

Table A.2 gives the round trip communication times and the derived one-way message passing rates.

The communication rate is plotted against the message size in Figure 20. The asymptotic interprocessor communication rate is 225 kbytes per second for one-way, nearest-neighbor message passing. The node-to-host communication rate was calculated in a similar way and found to be 75 kbytes per second. Further, these results indicate that the net com-

Table A.1. Floating Point Benchmarks

Operation	Milliseconds per 10^5 operations	Operations per second
$R*R$	7728	12940
$R*R*R$	10304	9705
$R+R$	6624	15087
R/R	8144	12278
$\sqrt{R}$	7440	13440
exp(R)	37056	2699
do-loop	2528	—

Table A.2. Node-to-Node Communication Rates

Message size (bytes)	Iterations	Time (ms)	Kbyte/s
4	100	0.90	0.89
40	100	1.07	7.46
1000	300	4.32	138.9
1040	300	5.52	113.0
2040	300	5.87	208.4
2040	300	8.26	151.2
4080	300	12.26	199.7
4120	300	13.20	187.3
8160	300	22.64	216.3
8200	300	24.35	202.0
12280	300	33.39	220.7
12320	300	35.01	211.2
16000	200	42.91	223.7

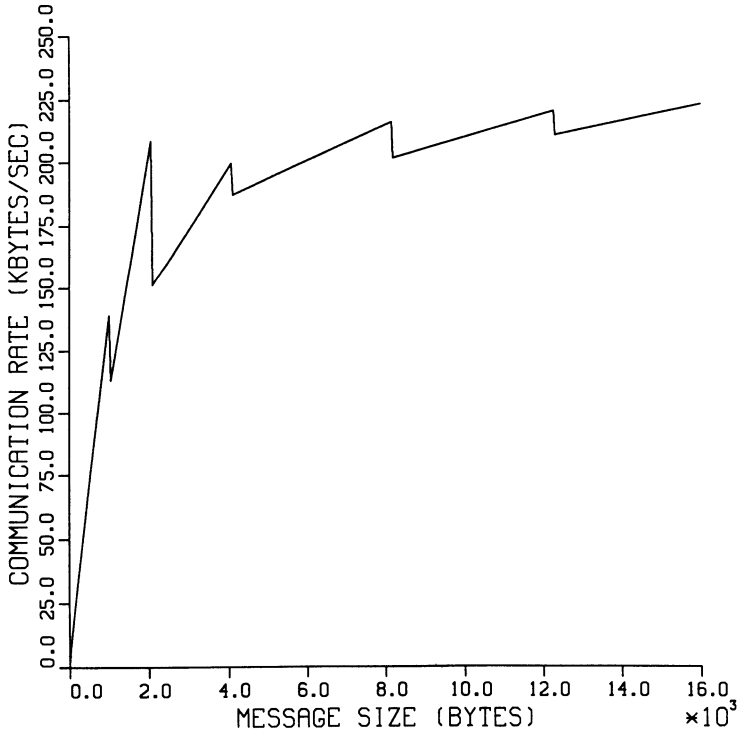


Figure 20. Node-to-node communication rate versus message size.

munication rate is strongly dependent on the message size and that the messages are broadcast in 1-kbyte chunks. Hence, when developing algorithms for the Hypercube, communication overhead can be minimized by avoiding small messages or by bundling them together into larger blocks. Comparing the node-processing speed of 19,230 floating point operations per second with the maximum communication rate indicates that about 1.5 double precision numbers can be passed between processors in the time that one operation can be carried out.

Appendix B: Benchmark Test Cases

The parameters for the two benchmark test molecules are given in Tables B.1 and B.2. Note that the basis sets are fully uncontracted to simplify load balancing.

Table B.1. Parameters for C_2H_2

Geometry

C–C	1.50 Bohr
C–H	0.75 Bohr

Energies

SCF	−112.464174140 hartrees
Moller–Plesset ($E^{(2)}$)	−0.108571920639 hartrees

Basis sets

Carbon			Hydrogen		
No.	Angular momentum	Exponent	No.	Angular momentum	Exponent
1	S	42.4974	1	S	2.8992
2	S	14.1892	2	S	0.6534
3	S	5.1477	3	S	0.1776
4	S	1.9666			
5	S	0.4962			
6	S	0.1533			
7–9	P	0.54424			

Table B.2. Parameters for H_2O

Geometry	
O–H	1.8523498 Bohr
H–O–H	104.0330035 degrees

Energies	
SCF	−63.195575507070 hartrees
Moller–Plesset ($E^{(2)}$)	−0.0694749326263 hartrees

Basis sets

Oxygen			Hydrogen		
No.	Angular momentum	Exponent	No.	Angular momentum	Exponent
1	S	5.0	1	S	0.5
2	S	1.0	2	S	1.0
3	S	0.5			
4–6	P	0.75			
7–9	P	1.25			

References and Notes

1. J. Von Neumann, transcript of talk given at the Institute for Advanced Study, May 15, 1946, Office of Naval Research, Washington D.C.
2. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, *J. Chem. Phys.* **21**, 1087 (1953).
3. S. F. Boys, G. B. Cook, C. M. Reeves, and I. Shavitt, *Nature* **178**, 1207 (1956).
4. From the Cray-1 at 12.5 ns in 1976 to the Cray-2 at 4.1 ns in 1985.
5. B. L. Buzbee, Technical Report No. LA-UR-83-1389, Los Alamos National Laboratories (1983).
6. D. L. Slotnick, W. C. Borck, and R. McReynolds, *Proc. Fall Jt. Computing Conf. AFIPS Conf. Proc.* **22**, 97 (1962).
7. G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. J. Slotnick, and R. A. Stokes, *IEEE Trans. Comput.* **C-17**, 746 (1968).
8. Many references in Y. Paker, *Multi-Microprocessors*, Academic Press, New York (1983).
9. G. Fiellant and D. Rogers, *Electronic Design* **32**(18), 153–168.
10. FPS T Series, Sales Brochure from Floating Point Systems, Inc., Box 23489 Portland, Oregon 97223.
11. U. Bernutat-Buchmann, D. Rudolph, and K. H. Scholtter, *Parallel Computing Eine Bibliographie*, University of Bochum, Bochum (1983).
12. N. S. Ostlund, R. A. Whiteside, and P. G. Hibbard, *J. Phys. Chem.* **86**, 2190 (1982).
13. E. Clementi, G. Corongiu, J. H. Detrich, L. Domingo, A. Laaksonen, H. L. Nguyen, and S. Chin, Technical Report No. POK-40, IBM (1984).

14. S. Otto, Caltech/JPL Concurrent Computing Project Annual Report 1983–1984, Vol. 2, Applications, Caltech (1985).
15. E. A. Hylleraas, *Z. Phys.* **48**, 469 (1928).
16. H. M. James and A. S. Coolidge, *J. Chem. Phys.* **1**, 825 (1933).
17. J. H. Van Vleck and A. Sherman, *Rev. Mod. Phys.* **7**, 167 (1935).
18. R. S. Mulliken and C. C. J. Roothan, *Proc. Natl. Acad. Sci. U.S.* **45**, 395 (1959).
19. Many examples in H. F. Schaefer III, *Science* **231**, 1100 (1986).
20. E. Clementi, G. Corongiu, J. H. Detrich, S. Chin, and L. Domingo, *Int. J. Quantum Chem. Symp.* **18**, 601 (1984).
21. M. Dupuis, private communication.
22. For examples where relativistic effects are significant, see P. A. Christiansen, W. C. Ermler, and K. S. Pitzer, *Ann. Rev. Phys. Chem.* **36**, 407 (1985).
23. B. T. Sutcliffe, in *Computational Techniques in Quantum Chemistry* (G. H. F. Diercksen, B. T. Sutcliffe, and A. Viellard, eds.), Reidel, Boston (1975), p. 1.
24. A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, Macmillan, New York (1982).
25. I. Shavitt, in *Methods of Electronic Structure Theory* (H. F. Schaefer III, ed.), Plenum Press, New York (1977), p. 189.
26. J. A. Pople, J. S. Binkley, and R. Seeger, *Int. J. Quantum Chem. Symp.* **10**, 1 (1976).
27. R. J. Bartlett, *Ann. Rev. Phys. Chem.* **32**, 359 (1981).
28. C. F. Bender, *J. Comp. Phys.* **9**, 547 (1972).
29. Y. Osamura, Y. Yamaguchi, P. Saxe, M. A. Vincent, J. F. Gaw, and H. F. Schaefer III, *Chem. Phys.* **72**, 131 (1982).
30. J. E. Rice, R. D. Amos, N. C. Handy, T. J. Lee, and H. F. Schaefer III, *J. Chem. Phys.* **85**, 963 (1986).
31. J. A. Pople, R. Krishnan, H. B. Schlegal, and J. S. Binkley, *Int. J. Quantum Chem. Symp.* **13**, 225 (1979).
32. Responses to First Survey of Parallel Processing Projects, compiled by Arvind, Laboratory for Computer Science, MIT (1985).
33. W. D. Hillis, *The Connection Machine*, MIT, Cambridge (1985).
34. D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, *Science* **231**, 967 (1986).
35. J. E. Brandenburg and D. S. Scott, Technical Report No. 280182-001, Intel Scientific Computers (1983).
36. E. Felton, S. Karlin, and S. Otto, Caltech/JPL Concurrent Computing Project Memo 92B.
37. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP Completeness*, W. H. Freeman, San Francisco (1979).
38. Review listing many references: D. Hegarty and G. Van der Velde, *Int. J. Quantum Chem.* **23**, 1135 (1983).
39. S. Obara and A. Saika, *J. Chem. Phys.* **84**, 3963 (1986).
40. H. Taketa, S. Huzinaga, and K. O-ohata, *J. Phys. Soc. Jpn.* **21**, 2313 (1966).
41. B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, New Jersey (1980).
42. C. G. J. Jacobi, *J. Reine Angew. Math.* **30**, 51 (1846).
43. A. H. Sameh, *Math. Comp.* **25**, 579 (1971).
44. R. A. Whiteside, N. S. Ostlund, and P. G. Hibbard, *IEEE Trans. Comput.* **C-33**, 409 (1984).
45. R. P. Brent and F. T. Luk, *SIAM J. Sci. Stat. Comp.* **6**, 69 (1985).
46. V. R. Saunders and J. H. van Lenthe, *Mol. Phys.* **48**, 923 (1983).
47. H. B. Schlegel, *J. Chem. Phys.* **84**, 4530 (1986).
48. E. R. Davidson, *J. Comp. Phys.* **17**, 87 (1975).

49. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts (1980), pp. 264–270.
50. P. J. Reynolds, D. M. Ceperley, B. J. Alder, and W. A. Lester, *J. Chem. Phys.* **77**, 5593 (1982).
51. R. A. Friesner, *Chem. Phys. Lett.* **116**, 39 (1985).
52. R. A. Friesner, *J. Chem. Phys.* **85**, 1462 (1986).
53. S. Eigenback and C. Sadler, *BYTE*, p. 181 (Aug. 1985).
54. T. Toffoli, *Physica* **10D**, 117 (1984).
55. U. Frisch, B. Hasslacher, and Y. Pomeau, *Phys. Rev. Lett.* **56**, 1505 (1986).
56. Technical Report No. 86.14, Thinking Machines Inc., Cambridge (1986).
57. S. Wolfram, *J. Stat. Phys.* **45**, 471 (1986).
58. P. Clavin, D. d'Humieres, P. Lallemand, and Y. Pomeau, *C. R. Acad. Sci. Paris*, t. 303, Series II, No. 13, pp. 1169–1174 (1986).
59. D. Montgomery and G. D. Doolen, Technical Report No. LA-UR-86-2975, Los Alamos National Laboratories (1986).

Contents of Previous Volumes

Volume 1: Electron Correlation in Atoms and Molecules

1. **Electron Correlation in Atoms**
   Karol Jankowski
2. **Electron Correlation in Molecules**
   Miroslav Urban, Ivan Černušák, Vladimír Kellö, and Jozef Noga
3. **Four-Index Transformations**
   Stephen Wilson
4. **Green's Function Monte Carlo Methods**
   B. H. Wells

Volume 2: Relativistic Effects in Atoms and Molecules

1. **Relativistic Atomic Structure Calculations**
   Ian P. Grant
2. **Relativistic Molecular Structure Calculations**
   Stephen Wilson
3. **The Relativistic Effective Core Potential Method**
   Odd Gropen
4. **Semiempirical Relativistic Molecular Structure Calculations**
   Pekka Pyykkő
5. **Relativistic Many-Body Perturbation Theory**
   Harry M. Quiney

Author Index

Subject Index