Ab initio Quantum Chemistry in Parallel - Portable Tools and Applications

Robert J. Harrison, Ron Shepard,[1]
Theoretical Chemistry Group, Chemistry Division,
Argonne National Laboratory, Argonne IL 60439, U.S.A.

Rick A. Kendall,[2][3]
Theory, Modeling and Simulation Program,
Mail-Stop K2-18, Molecular Science Research Center,
Battelle Pacific Northwest Laboratories, Richland, WA 99352.

In common with many of the computational sciences, *ab initio* chemistry faces computational constraints to which a partial solution is offered by the prospect of highly parallel computers. *Ab initio* codes are large and complex ($O(10^5)$ lines of FORTRAN), representing a significant investment of communal effort. The often conflicting requirements of portability and efficiency have been successfully resolved on vector computers by reliance on matrix oriented kernels. This proves inadequate even upon closely-coupled shared-memory parallel machines. We examine the algorithms employed during a typical sequence of calculations. Then we investigate how efficient portable parallel implementations may be derived, including the complex multi-reference singles and doubles configuration interaction algorithm. A portable toolkit, modeled after the Intel iPSC and the ANL-ACRF PARMACS, is developed, using shared memory and TCP/IP sockets. The toolkit is used as an initial platform for programs portable between LANS, Crays and true distributed-memory MIMD machines. Timings are presented.

[1]E-mail: harrison@tcg.anl.gov. shepard@tcg.anl.gov. Tel: (708) 972-7201.

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

[3]E-mail: d3e129@pnlg.pnl.gov. Tel: (509) 375-2602.

1 Introduction

Ab initio chemistry has as its ultimate goal the determination of all chemical information from quantum mechanics with reference to only the fundamental constants as experimental input. More realistically, theoretical calculations serve as a useful complement to experiment, providing both qualitative interpretation and quantitative results which may guide experiments and augment their results.

In practice, most of the chemistry of the first two rows in the periodic table may be determined from the time independent non-relativistic Schrödinger equation, a second-order linear differential equation for the wavefunction describing the molecule or system of interest. The wavefunction is a function of the coordinates of the electrons and nuclei in the molecule. Due to the mass difference of electrons and nuclei, this equation approximately separates (the Born-Oppenheimer approximation), leading to an equation for the electronic wavefunction at a specific nuclear geometry. The electronic equation is then (in atomic units)

$$\left(-\frac{1}{2}\sum_i \nabla_i^2 - \sum_{i,\mu}\frac{Z_\mu}{R_{i\mu}} + \sum_{i>j}\frac{1}{r_{ij}} + V\right)\psi_\lambda = \mathbf{H}\psi_\lambda = E_\lambda\psi_\lambda \tag{1}$$

where i labels electrons, μ labels nuclei, Z_μ is a nuclear charge and $R_{i\mu}$ and r_{ij} are interparticle distances. External potentials (e.g. an applied electric field) are represented by V. E_λ is the λ^{th} eigenvalue or energy and ψ_λ the corresponding eigenfunction or wavefunction. The wavefunction is now an explicit function of the electronic coordinates and an implicit function of those of the nuclei.

Solution of the electronic problem at many nuclear geometries provides an effective potential energy surface upon which the nuclei move. One may then solve for the nuclear motion and, if required, include coupling between multiple electronic states due to the only approximate separability of electronic and nuclear motion. In the following we shall consider only the solution of the Born-Oppenheimer electronic problem. The nuclear motion problem is possibly yet more computationally demanding and is the subject of much theoretical investigation.

The computational bottleneck faced by ab initio *chemistry is acute. Possibly thousands of electronic structure calculations must be performed to define the potential energy surface. Each calculation scales as roughly the sixth power of the dimension of an underlying basis set, the size of which limits the achievable accuracy. Doubling the size of molecules at the limit of our current ability would result in doubling the dimensionality of the potential energy surface and increasing the expense of each electronic structure calculation by $O(2^6)$. Theoretical advances offer the only true solution, but they do not seem forthcoming. Increased computer power through massive parallelism seems to be the only route forward at present.*

To move forward we must then identify appropriate computer architectures for performing our calculations and resolve the ensuing algorithmic and software issues. In this paper we propose a consistent strategy for both migrating our existing software into a parallel environment and for the development of new *ab initio* codes explictly targeted at highly parallel machines. Portability and efficiency are prime concerns. Our attention is restricted to multiple instruction multiple data (MIMD) architectures, both shared and distributed memory. We present results to illustrate the performance achieved using a message-passing toolkit developed by us.

In Section 2, we present a minimum of the theory and terminology necessary to understand the algorithms employed and to appreciate at least some of their complexity. Section 3 considers some previous work on parallel versions of these algorithms. In Section 4, we turn our attention to arriving at implementations that are both efficient and portable, at least between certain classes of machines. Section 5 describes a message-passing toolkit that we have developed, which we are using as an initial platform for migration of our codes into a parallel environment. Finally, in Section 6, we discuss parallel implementations of several algorithms, and present timings on a variety of machines for *ab initio* production calculations.

2 Theory and Algorithm

2.1 Background

The exact solution available for the hydrogen atom (one electron) serves as the model for main-stream basis-expansion techniques which have proved very successful. The hyrogenic eigenfunctions are products of radial functions (exponentials times Laguerre polynomials) and angular functions (spherical harmonics). In molecular calculations, one-particle basis sets are typically chosen as atom-centered hydrogen-like functions (or atomic orbitals, AOs), except that the radial part is chosen as a linear combination of gaussians to faciliate the evaluation of the required integrals [1]. This basis expansion encapsulates the essential ingredients of chemistry (i.e. atoms in a molecular environment) and allows one to focus on the important outer, or valence region and apply cruder approximations to the chemically less important region near the nucleus.

Since electrons are fermions, the N-electron functions must be antisymmetric with respect to exchange of any pair of electronic coordinates. The simplest N-electron wavefunction in common use is a single antisymmetric product (or "Slater-determinant") of one-electron functions (or molecular orbitals, MOs) which are orthogonal linear combinations of the AOs. The expansion coefficients are determined by minimization of the energy in the Self Consistent Field (SCF) approach. This simplest of wavefunctions determines many properties with suprising accuracy and recovers over 99% of the total energy. However, chemistry is determined by small energy differences and the remaining 1% proves to be sensitive to processes such as bond formation and breaking which are not described well within the single-determinant approximation.

There are many methods to proceed beyond the SCF approximation. We shall focus on one which is currently the most accurate and widely applicable (multi-reference single- and double-execitation configuration interaction, MRSDCI) and discuss a new algorithm for producing accurate, but expensive, benchmark results (selected configuration interaction plus perturbation theory, CI+PT).

The SCF procedure defines a set of occupied orbitals and a complementary set of unoccupied orbitals. Within the limits of the one-particle basis, the exact solution may be written as a linear combination of all determinants (or configurations) formed by exciting 1, 2, ..., N electrons from the occupied set into the unoccupied set. This is the so-called full-CI wavefunction, a problem which grows roughly as the dimension of the AO basis to the power of the number of electrons. In practice, only a few excited configurations are very important, typically only involving valence orbitals. The expansion coefficients of these configurations and the MOs are simultaneously determined in the multiconfiguration SCF (MCSCF) procedure [2]. Finally, perturbation theory suggests that all single and double excitations from this MCSCF reference space will include the next most important contribution to the wavefunction. Taking this as the final expansion space, the linear expansion coefficients (c_I^λ) are determined by solving the large sparse (typically 2-3% dense) eigenvalue problem

$$\sum_I H_{IJ} c_J^\lambda = E^\lambda c_I^\lambda, \qquad (2)$$

where $H_{IJ} = < I|\mathbf{H}|J >$ is the matrix representation of the Hamiltonian operator (1) in the configuration space. The dimension of the expansion can easily be several million, and the largest used to date has been about 10^9 [3] (a full-CI calculation). Fortunately, we only need the lowest few eigenvectors and eigenvalues. However, we must be able to efficiently compute Hamiltonian matrix-vector products without storing the matrix. Much theoretical work was required before this very general form of wavefunction could be routinely applied [4, 5, 6, 7, 8]. The algorithm is now dominated by small (O(10-100)) matrix-matrix and matrix-vector products and, on a single CRAY-YMP cpu, can sustain over 200 MFLOPS for hour after hour, including the overhead of prodigous amounts of I/O [9, 10].

In order to assess the error inherent in the MRSDCI method, we need the exact (i.e. full-CI) result. For most systems of interest we can only approximate this result, and a new method [11], developed by one of us (RJH), permits the systematic approximation of the full-CI result. The reference configurations are chosen according to some interaction threshold (T) and this space is treated exactly. Second-order perturbation theory is used to partially treat the negelected space. As T approaches zero, the model converges to the full-CI result.

Before we can start explicit consideration of parallel algorithms, we must look in just a bit more detail at each step of the calculation.

2.2 AO Integral Evaluation

The Hamiltonian (1) is a two-particle operator, and matrix elements between two N-electron functions which are products of one-electron functions are expressible as sums of one- and two-electron integrals. Thus the first step is to compute these integrals over the AO basis (χ_μ):

$$\begin{aligned} h_{\mu\nu} &= <\mu| -\frac{1}{2}\nabla^2 - \frac{Z_\mu}{R_{i\mu}} + V|\nu> \\ &= \int d\tau_1\, \chi_\mu(\vec{r_1})(-\frac{1}{2}\nabla^2 - \frac{Z_\mu}{R_{i\mu}} + V)\chi_\nu(\vec{r_1}) \qquad (3) \\ (\mu\nu|\sigma\lambda) &= <\mu\lambda|r_{12}^{-1}|\sigma\nu> \\ &= \int d\tau_1\, d\tau_2\, \chi_\mu(\vec{r_1})\chi_\nu(\vec{r_1})\frac{1}{r_{12}}\chi_\sigma(\vec{r_2})\chi_\lambda(\vec{r_2}) \qquad (4) \end{aligned}$$

Thanks to much theoretical and algorithmic development [12, 13, 14, 15, 16, 17] integral evaulation is very efficient. The number of two-electron integrals (4) scales as $O(M^4)$, M being the dimension of the AO basis. These integrals are usually computed and stored on disk, exploiting permutational symmetry of the integral labels and any sparsity. Typical basis dimensions are between 50 and 250, though dimensions of 1,000 or more have been used. Algorithms are now being employed which compute these integrals as required, trading I/O storage limits for increased computation requirements [18]. The important element to note is that each integral may be evaluated independently. In practice it is far more efficient to evaluate batches of integrals involving the same set of radial functions.

2.3 SCF Algorithm

The SCF algorithm [19, 20] is an iterative process requiring the construction at each stage of a Fock matrix (F) from an approximate density matrix (P) and the list of AO one- and two-electron integrals. For closed shell (only doubly occupied orbitals) systems this is simply

$$F_{\mu\nu} = h_{\mu\nu} + \sum_{\sigma\lambda} P_{\sigma\lambda}[(\mu\nu|\sigma\lambda) - \frac{1}{2}(\mu\lambda|\sigma\nu)]. \tag{5}$$

The Fock matrix is then transformed into the MO basis, diagonalized, and the density matrix updated. Since we store only the permutationally unique integrals, each integral we process may actually contribute to up to six elements of the Fock matrix. The Fock matrix construction scales as $O(M^4)$ and the transformations and diagonalization as M^3. $O(M^4)$ I/O is performed upon the AO integrals.

2.4 Four-Index Transformation

Although it is possible to express the MRSDCI equations in the AO basis, it is most straightforward to work in the MO basis. This requires a transformation [21, 22, 23] of the one- and two-electron integrals

$$\begin{aligned} h_{ij} &= \sum_{\mu\nu} C_{\mu i} C_{\nu j} h_{\mu\nu} && (6) \\ (ij|kl) &= \sum_{\mu\nu\sigma\lambda} C_{\mu i} C_{\nu j} C_{\sigma k} C_{\lambda l} (\mu\nu|\sigma\lambda), && (7) \end{aligned}$$

where Greek indices are used to label AOs, Roman indices for MOs. and $C_{\mu i}$ is the coefficient of the μ^{th} AO in the i^{th} MO. The transformation of the two-electron integrals is an $O(M^5)$ process, expressible as a sequence of matrix products of dimension M. The peak speed of many vector processors is thus readily sustained throughout the transformation. Note that I/O of $O(M^4)$ must be performed on the AO and MO integrals and required intermediates.

2.5 MRSDCI

The MRSDCI calculation is equivalent to extraction of a few lowest eigenvalues and vectors from the sparse matrix representation of the Hamiltonian in the CI expansion space. This proceeds iteratively via construction of the action of the Hamiltonian matrix (H) on some trial vector (c)

$$\begin{aligned} \sigma_I &= \sum_J H_{IJ} c_J \\ &= \sum_{ijJ} A_{ij}^{IJ} h_{ij} c_J + \frac{1}{2} \sum_{ijklJ} A_{ijkl}^{IJ} (ij|kl) c_J, \end{aligned} \tag{8}$$

where capitalized Roman indices label configurations. The coupling coefficients A_{ij}^{IJ} and A_{ijkl}^{IJ} are extremely sparse. The theory underlying their efficient evaluation and factorization involves graphical representations of the unitary and symmetric groups [5, 6, 7, 8]. Pre-sorted integrals are read in large blocks, and their contribution accumulated. This is driven by the type of each integral [4, 6, 23] (the number of orbital indices occupied in the reference space) and then by a list of precomputed formulae for model coupling coefficients. A matrix-matrix or matrix-vector product is then used to multiply the integrals and CI vector (c) for related interactions with a common coupling coefficient. The c and σ vectors may be explicitly paged from disk, though there is overhead associated with this. The integrals are read once each iteration. The computation scales roughly as $O(M^6)$, with $O(M^4)$ I/O on the integrals. Note that the algorithm is complex and that it is possible to process required data (c, σ and the integrals) in pages.

2.6 Selected CI and Perturbation Theory

This code [11] is much simpler than the sophisticated MRSDCI, but produces benchmarks used to assess the accuracy of MRSDCI and other approximate models. We choose an initial reference space and selection threshold. Then all configurations which interact with the reference more than the threshold are added to the reference space. Then second-order perturbation theory is used to improve the accuracy of the energy.

$$\Delta E^{\lambda} = \sum_{I} \frac{|<I|\mathbf{H}|\lambda>|^2}{E^{\lambda} - <I|\mathbf{H}|I>} \tag{9}$$

where λ labels the eigen-state of interest and I labels configurations in the complement of the reference space. Both of these steps are combined into one for efficiency. The threshold is reduced and the whole procedure repeated until convergence is demonstrated. Since we have abandoned the regular structure of MRSDCI, we can no longer achieve the high vector processing rates. However, each term $|<I|\mathbf{H}|\lambda>|^2/(E^{\lambda} - <I|\mathbf{H}|I>)$ may be determined independently for each I (possibly $O(10^7 - 10^8)$ values), and the result collapsed at the end of the computation. This assumes random access to the MO integrals, which is reasonable for small basis sets.

2.7 Sample Calculation

To provide some perspective for how long each of these calculations takes in practice, we present cpu timings and average MFLOP ratings (on a single processor CRAY-YMP) for a high-accuracy calculation on CH_3. This used a cc-pvqz basis [24] (i.e. 145 functions), C_s symmetry, and a seven-electron seven-orbital complete active space MCSCF reference (764 CSF). The dimension of the final CI expansion was 5,666,940. The COLUMBUS program system [10] was employed.

- AO integrals - 22 minutes (13.6 million integrals), 9.1 MFLOPS

- MCSCF procedure – 71 minutes (5 iterations), 35.1 MFLOPS
- Four-index transformation – 8 minutes, 63.5 MFLOPS
- MRSDCI calculation – 200 minutes (12 iterations), 231 MFLOPS

Lower accuracy calculations tend to de-emphasize the MCSCF and MRSDCI steps. The AO integral program used is almost completely scalar. The current implementations of the MCSCF optimization and four-index use mostly vector, rather than matrix, operations. The MRSDCI is dominated by matrix multiplications. These characteristics partially account for the disparity between the scaling of the algorithms with basis set dimension and the observerd execution times. The remainder is due to the prefactors; e.g. approximate FLOP counts for the four-index and AO integral steps are roughly $2M^5$ and $10^2 - 10^4 M^4$ respectively.

3 Previous Parallel Implementations

We include here a brief, and of necessity, incomplete, review of previous work. High accuracy *ab initio* codes, typified by MRSDCI, have had great success upon conventional vector super computers, but we shall not discuss that here. Work on parallel algorithms falls fairly neatly into three loose categories. First, few-processor shared-memory machines, typified by Alliant, CRAY-XMP and CRAY-2 systems. Second, few-processor distributed-memory machines, characterized by the IBM LCAP systems. Third, many-processor distributed-memory machines, including the first three generations of the Intel iPSC.

3.1 AO Integral Evaluation

The highly parallel nature of this algorithm makes this the first that most people attack, usually by having each process compute some subset of the integrals. There have been many successful implementations (e.g. IBM LCAP [25, 26, 27, 28, 29, 30], CRAY [31], hypercube [32, 33, 34]) which suggest that linear scaling up to hundreds of processors is possible. However, even for this simple test case there have been cautions. The first work on the Intel iPSC-1 [32, 33] suffered from too little local memory and insufficient interprocessor bandwidth in the subsequent SCF calculation. The paucity of usable local memory (320Kbytes [32]) forced the adoption [32] of a very compact integral program which computed integrals one at a time rather than in batches. We estimate this to degrade performance by one or more orders of magnitude for high angular momentum basis functions. However, good scaling was observed. LCAP applications with very large systems in small basis sets and small systems in large basis sets suggest that good linear scaling is achieved with a deterministic decomposition of the work [28, 30], whereas our work on small systems with large basis sets revealed that load balancing was essential [35]. This discrepancy is probably due to differing use of spatial symmetry and our use of generally-contracted basis sets with high angular momentum functions. We have been the only group so far to address the issue of portability with our previous distributed-memory implementation for shared-memory UNIX machines [35]. This work suffered from an I/O bottleneck when concatenating each process's integral file at the end of the calculation, for subsequent use by a serial SCF program.

No work has been reported on single instruction multiple data (SIMD) machines to date, even though this holds some promise. Most of the algorithms that use the integrals do not map well onto the SIMD model.

3.2 SCF Algorithm

Here more mixed success has been reported [32]. Simple replication of the Fock matrices (5) and partitioning of the integral list (as done automatically by a parallel integral program) is all that is required to parallelize the $O(M^4)$ Fock matrix construction (e.g. IBM LCAP [25, 26, 27, 28, 29, 30], CRAY [36], hypercube [32, 33]).

However, this assumes sufficient local memory to hold at least two $O(M^2)$ matrices, and additional local memory or disk for the integrals. Since only 12 floating point operations and some non-linear addressing is done per integral, it is easy to become I/O limited [36]. All researchers note that the $O(M^3)$ steps start to dominate, except for very large problems or on few-processor powerful machines where the diagonalization and transformations are highly optimized [29, 32, 36]. Guest [34] has distributed the matrix transformations, and the Sandia group has reportedly [37] seen linear scaling up to 128 processors on an iPSC-2, and so must have parallelized all the $O(M^3)$ steps. Matrix diagonalization on parallel machines remains an active research area [38, 39, 40]

3.3 Four-Index Transformation

Although this is perhaps the most well structured algorithm of those discussed here, this transformation has proven a formidable problem on current generation parallel machines. Few processor, shared-memory algorithms have been proposed [41, 42]. The first of these failed due to the assumption that all information was memory resident and saturated available memory bandwidth with SAXPY operations. As expected (on an Alliant FX/8 and even a CRAY-XMP) only matrix-multiply based algorithms avoid the memory contention suffered by naive parallel-vector algorithms [42]. The second algorithm [42] failed to parallelized the required sorts/transpositions. Transformation algorithms on the IBM LCAP systems also suffered from bottlenecks in the sorting stages [29, 43, 44], alleviated only by the provision of shared memory [43, 44]. Again, the first work [32, 45] on an inappropriately configured iPSC-1 was forced into unfortunate algorithmic compromises by the small memory and slow communications. Subsequent development [46, 47] and work on the iPSC-2 and ncube-2 [34, 37] suggests that good scaling with an efficient algorithm can be achieved with a balance of local memory, disk bandwidth, processor speed and interprocessor bandwidth. However, no hard performance data has been published to demonstrate this.

3.4 MRSDCI

Most work has suffered from the lack of an efficient four-index transformation [43, 44] and has been limited to few-processor distributed-memory machines [29]. The most successful decomposition, on an LCAP-1 system at ECSEC [29], gave each processor a complete copy of all the data sets, barring the formula list for the coupling coefficients, which was split deterministically between the processors. Thus, the $O(M^6)$ step was parallelized, but the total amount of local memory and disk increased linearly with the number of processors. In a simulation of four processes on an IBM-4383, a speed-up of only 2.6 was achieved due to poor distribution of the load, rather than overhead.

4 Portability and Efficiency

In an ideal world, one could code an algorithm in an appropriate language to run on the appropriate architecture for efficient execution. Portability would be achieved by compilers simulating the model architecture within the class of machines that supported an efficient simulation (e.g. MIMD simulating SIMD, shared-memory MIMD simulating distributed-memory MIMD). A quick glance at the current state of parallel languages and programming environments brings one down to earth with a bump. In the area of high performance scientific computing, one basically has to accept FORTRAN with extensions. This is not so bad, as FORTRAN is still the best language for expressing numerically intensive formulae. It also provides backward compatibilty with our large collection of dusty- and not-so-dusty-deck FORTRAN programs. FORTRAN is, of course, deficient in all other respects [48].

The use of optimized BLAS (notably matrix multiply) on shared-memory computers proves insufficient even for algorithms dominated by these operations, such as

MRSDCI. Even very large calculations result in only matrix dimensions $O(10-100)$. On an Alliant FX/8, matrix dimensions of around 60 are required for full efficieny, the corresponding number for a multiprocessor Cray being well over 200. Compiler-based loop parallelism is actually very portable between shared-memory computers (e.g. Cray, Alliant, Convex, Stardent) but proves inadequate for an efficient implementation of most quantum chemistry applications without an extensive re-working of the algorithms to make a large-enough grain parallelism visible to the compiler. Naive parallel vector algorithms suffer due to memory bandwidth problems. Finally, if we are to go to the effort of re-coding, we wish a bigger payoff than the limited scaling offered by shared-memory machines.

It should be apparent that only AO integral evaluation (and perhaps SCF and four-index) might be a good match for a SIMD machine and at least limited success on distributed-memory machines has been achieved for all the steps. Thus we target distributed-memory MIMD machines, encompassing shared-memory machines through efficient simulation. The volume of data processed by our algorithms places rather large lower bounds upon the local memory, disk and inter-processor bandwidths. Specifically,

- Local memory and disk space should be equivalent to a total of $O(10)$Gb to hold integrals etc.;

- Disk and inter processor bandwidth relative to cpu speed may be estimated from Cray data, where the SCF and four-index are not I/O bound with at least 6-10 Mb/s sustained bandwidth to disk;

- The desired level of granularity suggests that the minimum amount of local memory should be 4-16Mb per node, with the MRSDCI code requiring more memory than the integral and SCF programs.

The availability in early 1991 of the Delta Touchstone prototype suggests that these requirements can be met. We also have an appropriately configured MIMD available to us now in the form of workstations and mini-supers on our LAN.

For deciding which portable parallel programming environment or tools to adopt, we had the following criteria:

- FORTRAN must be fully supported, including FORTRAN I/O;
- It must be available and supported on a wide variety of platforms, including most UNIX workstations, mini-supercomputers, supercomputers, networks of these, and true distributed-memory machines such as ncube and Intel; and
- It must be affordable by both us and the wide community of developers and consumers of our codes.

We examined in depth STRAND [49], Linda [50], and PARMACS [51], and discussed the issue with the Advanced Computing Reseach Facility (ACRF) at ANL. Linda [50] seemed very attractive, but FORTRAN Linda was not then a supported product, network Linda was in alpha test, and in any case we could not afford to purchase the required number of versions. STRAND [49] was also too expensive and was not availble on many interesting platforms. The PARMACS [51] are a set of m4 macros which implement a portable message passing model. These tools were low-level, fine for small tests, but did not support the FORTRAN runtime environment, were not robust and had no support at all.

The explicit message-passing model seemed the most natural way of expressing the algorithms, especially since we were concerned about efficient inter-process communication and local memory usage. Also, the analysis required to arrive at, for instance, a Linda implementation of the integrals and SCF was essentially identical to that for the message passing, both being data-driven parallel. Thus the choice of low-level message passing seemed to also permit ready migration to a higher-level environment at a later date. We were lead then to implementing a replacement for the PARMACS, which we discuss next.

5 Message-Passing Toolkit

Since we would have to support this toolkit ourselves, we chose to implement the minimal set of operations required by our current algorithms, but with hooks for increased functionallity if required. A decision was taken to support the toolkit only on either true message-passing machines, or UNIX-based computers. In practice, some mechanism of process creation and an equivalent to TCP/IP sockets (e.g. VAX/VMS mailboxes) is the minimum required. We combined the ANL PARMACS [51] macro interface with that of the Intel iPSC to arrive at our subroutine interface, with the intent that the toolkit would be a very thin software layer on the iPSC and ncube. We are in the process of porting this to the iPSC and ncube. Some functionality was added, compared to the PARMACS [51], notably simultaneous support of shared memory and sockets, support of machines with differing number representations, support for FORTRAN I/O, and a standard load-balancing mechanism.

For ease of implementation, there is a single, untyped channel between each process, with strongly-typed messages received in the order sent. The strong typing resolves problems due to messages sent/received out of order, and provides a hook for translation between machines with different number representation. Several type values are reserved by the system for this purpose. For instance, with the correct type, a message comprising an array of integers could be successfully sent from a Sun to a Cray without the programmer being concerned with, or aware of, the translation. More commonly, byte ordering and big/little-endian issues must be confronted. Each message is tagged by the ID of the originating process and the counter value for the number messages sent to the receiving proceess. This facilitates debugging when message tracing is enabled.

Identical processes on the same computer communicate via shared memory (if supported by the O/S), using vector hardware if present. Otherwise, TCP/IP sockets are used. On a UNICOS Cray, with both processes memory resident, socket and pipe communication actually takes place via copying from buffer to buffer. Semaphores are used to synchronize shared-memory access, rather than spin-locks, as these tools will be used on some machines where time is literally money. There is some performance penalty associated with this, so the semaphores may be replaced in sections where communicating processes are sychronized and ready to exchange data.

The load-balancing mechanism implements a simple shared counter for a single, active loop being executed by some subset of the processes. Processes block at the end of the loop to avoid overlapping of active loops. This is sufficient for us to implement the coarse-grain load balancing required for the integrals and CI+PT algorithms, for which results will be presented. On the iPSC and ncube, this requires dedication of one or more processors to this service. This is deemed reasonable, as even a modest 10% load balancing failure on 128 processors is equivalent to discarding 12.8 processors.

Parallel programs in the UNIX environment are invoked with the command **parallel** with a configuration file name as an argument (PROCGRP file in the parlance of PARMACS [51]). If no argument is presented, then various default options are explored. The file specifies user information, process placement, the number of copies of each process and the working directory for the process. With the same configuration file, exactly the same set of processes are invoked from anywhere on the network. The process running the command **parallel** is also responsible for managing the shared counter. The application program must call initialization and termination routines for the message-passing environment.

An incomplete list of the functionality provided is:

- **snd()/rcv()/brdcst()** – message-passing routines

- **synch()** – synchronize all processes

- **nnodes()/mynode()** – process information
- **setdbg()** – toggle debugging and tracing
- **stats()** – print communication statistics
- **nxtval()** – read and increment shared counter.

6 Results and Discussion

We will discuss initial implementations of the AO integral, CI+PT and MRSDCI algorithms, and analyze timings for sample calculations.

6.1 AO integrals

This readily-parallelized algorithm serves to illustrate an advantage of the data-driven distributed-memory approach we are taking. The program contains a four-fold nest of loops over the labels of the radial functions in the basis set. Within this loop nest a large subroutine call tree (approx. 10.000 lines of code) evaluates the desired batch integrals. Our program (ARGOS [10, 52]) is typical of most older programs in that information is passed through global variables as well as arguments. so that substantial modification is required for a shared-memory implementation. The use of a distributed-memory model sidesteps this problem completely, and results in no increased complexity (since there is no global transformation of the locally computed integrals, and all the integrals are independent). Indeed, only a few dozen lines of new code are required for a minimal port of the serial code to the message-passing toolkit.

Tables 1 and 2 present preliminary timings from runs on a small Sun SparcStation Ethernet LAN. Separate integral files were produced by each process. with just one process per processor. The first test (Table 1) scales well except for five processes.

This may be partly due to increasing overhead from load-balancing on too fine a scale (as suggested by the increase in total cpu time). The variation in the five cpu times (6%) is larger than expected with load balancing for this problem (which has a small dynamic range in task size), but this is complicated due to the timings for the one-electron integrals not being included. Also the fith Sun was running an different release of SUN O/S and FORTRAN. We have not had opportunity to explore this further. The second test (Table 2, the carbon dimer) does not scale so well, and apart from the afcre-mentioned problem with the fifth Sun. is explained as a load-balancing problem. The combination of high-angular momentum functions (up to L=4, or g), highly-contracted p functions and the equvialence of the two carbon atoms conspire to produce a large variation in the task size. This would not necessarily be a problem but the largest tasks arise at the end rather than the beginning! We are addressing this issue.

For backwards compatability the integral program is able to concatenate the partial integral files. either one the fly, or at the end of the computation. This substantially reduces efficiency, but allows us to port the programs one at a time without destroying their utility.

6.2 MRSDCI

Similar to the integral program. the parallel structure we wish to exploit in MRS-DCI is many levels of subroutine call above the actual computation. Unlike the integral program. we are forced to manipulate distributed data (unless we assume unlimted local memory and disk space). so the distributed-memory implementation adds complexity.

We note above (Section 2.5) that the MRSDCI program reads through the MO integrals sequentially in large blocks, and is able to page both the CI and σ vectors (8). These capabilities allow us to distribute the CI and σ vectors and the MO integrals between multiple processors, unlike previous work [29]. Each processor needs at least a block of the CI and σ vectors (ideally two of each to exploit Hamiltonian symmetry) and one block of the integrals. If space permitted, information could be replicated to increase data-locality.

At worst, a block of integrals makes contributions to every element of the σ vector (e.g. with all indices in the occupied space). The CI coefficents involved would have the same unoccupied orbital indices as the σ vector elements. At the other extreme are integrals with no indices in the occupied space. The σ and CI coefficients involved in these interactions each carry two of the integral indices as part of their label.

The blocking of the data provides a partitioning of the work for a parallel algorithm into $K_{CI}K_{\sigma}K_{int}$ tasks, where K_{CI}, K_{σ} and K_{int} are the number of blocks of CI vector, σ vector and integrals, respectively. One of *many* possible strategies is to assign unique blocks of the σ vector to each processor, spread the integral blocks amoung the processors, and duplicate the CI vector as much as remaining storage will allow. Assuming fully dense interactions, the amount of data transfer or I/O is approximately $2N_{CI} + K_{CI}N_{int}$ where N_{CI} and N_{int} are the number of CI coefficients and integrals respectively, and M is again the number of basis functions. Assuming the work done by each of p processes is approximately $N_{CI}M^2/p$ and $N_{CI} \simeq N_{int}$, the ratio of communication to computation is approximately pK_{CI}/M^2. This justifies the premium placed upon local memory to minimize the blocking (i.e. K_{CI}). Use of the full structure of the Hamiltonian can reduce this ratio by up to a factor of M^2.

We are just beginning to explore these ideas and our first crude efforts used shared files for interprocess communication, rather than the tools we have now. Nevertheless, for a test calculation on F_2 with 319,000 CSF, splitting the CI and σ vectors into seven segments, we obtained a speed up of 1.9 from two processes on our Alliant FX/8. Hardly a basis for predicting good linear speedup, but a good start! Tasks were defined by the segment pairs of CI and σ vectors (28 in all), and load balancing was essential.

To further muddy the waters, we have omitted mention of the formula tape so far. It seems simplest to do away with the tape altogether and recompute the formulae as required. For small reference spaces this is a small overhead, but for large reference spaces and small basis sets it could double the cost of an iteration.

6.3 CI+PT

As discussed above, evaluation of (9) is readily parallelized over the index I. The code to evalulate each element of the sum is complex, but much simpler than the sophisticated MRSDCI program. Load balancing is essential, except for very large cases for which the statistical fluctuations of load become small. The program's objective is to approximate full-CI results, typically for many electron systems in small basis sets, so it is not unreasonable to assume that we may hold all the integrals in core (typically, we have only $10^5 - 10^6$). An efficient parallel implementation will enable valuable benchmarks to be performed.

Timings on several machines on our LAN, for a small and a slightly larger calculation. are presented in Tables 3 and 4. The times for the 'tiny' calculation (Table 3) are dominated by serial overhead in the algorithm and overhead from distributing the data. The larger calculation (Table 4) is a small example of a real production calculation and shows much better scaling. The failure at higher numbers of processors on the Alliant FX/8 may be ascribed to one or more of:

- Thrashing of the shared cache due to random access to the integrals;
- Load balancing at too fine a grain resulting in excessive overhead or contention for the server process: or
- Load balancing at too large a grain resulting in poor load balancing.

Analyzing the eight processor calculation, the range of cpu time in the parallelized section is 1518-1642s, which suggests the third explanation as the prime source of error. Typical benchmarks require one to two orders of magnitude more computation, with essentially no increase in the serial overhead, so this application should efficiently exploit several hundred processors. With minor modification it will be possible to provide limited paging of the integrals.

7 Conclusions

We have described some of the important algorithms in use in mainstream *ab initio* quantum chemical calculations and emphasized the value of portability as well as efficiency. It may not be possible, or even desirable, to have an efficient, single implementation portable between a two-processor Cray and a 1,000-processor hypercube. However, there is a large range of machines between these extremes which may be encompassed with such a code. We are also well aware that each machine will require its own, idiosyncratic optimizations.

We have adopted a coarse-grain, load-balanced, data-driven, message-passing model for all the algorithms discussed above. This may not be appropriate for other problems we encounter. The selection of low-level message passing was driven by consideration of cost and efficiency. In particular, it gives us explict control of valuable local memory and efficient use of interprocess communications. The development of the portable message-passing toolkit has provided us with an initial platform to develop our distributed-memory algorithms. We anticipate the adoption of higher level tools as they become more widely available and we gain more experience.

The MRSDCI calculation is currently the most expensive and complex step in the standard sequence of calculations. We have presented our inital thoughts on how to derive a parallel implementation from our efficient serial program system [10], along with very preliminary results. Much work is needed here, but we anticpate that with sufficient local memory, high efficiences will be achieved for large problems.

Finally, since theory is constantly moving forward we can anticipate new models supplanting the role of MRSDCI and requiring implementation on powerful, highly-parallel computers. Hopefully, software and hardware developments will bring this task more into line with the effort required to program a good, old-fashioned serial computer.

8 Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Basic Energy Sciences, Division of Chemical Sciences, under contract W-31-109-ENG-38. Work at the Pacific Northwest Laboratory was supported by the U.S. Department of Energy with the Battelle Memorial Institute under contract DE-AC06-76RLO 1830. Thanks are extended to J.K. Fisher for advice concerning this manuscript and to the staff of the ANL Advanced Computing Research Facility (notably Rick Stevens and Ian Foster) for many useful discussions and access to their computers. We are grateful to Rik Littlefield (PNL) for his insightful review of an early draft of this document, and Kim Ferris for use of his workstation.

References

[1] S.F. Boys, *Proc. Roy. Soc. Series A* **200** (1950) 542.

[2] R. Shepard in, *'Ab Initio Methods in Quantum Chemistry - II,'*ed. K.P. Lawley (John Wiley and Sons Ltd., 1987).

[3] J. Olsen, P. Jørgensen and J. Simons, *Chem. Phys. Letters* (1990), in press.

[4] B.O. Roos, *Chem. Phys. Letters* **15** (1972) 153.

[5] J. Paldus, *J. Chem. Phys.* **61** (1974) 5321; *Int. J. Quant. Chem.* **S9** (1975) 165; *Phys. Rev.* **A14** (1976) 1620.

[6] P.E.M. Siegbahn, *J. Chem. Phys.* **72** (1980) 1647.

[7] I. Shavitt, *Int. J. Quant. Chem.* **S11** (1977) 131; **S12** (1978) 5; *Chem. Phys. Lett.* **63** (1979) 421.

[8] *'Lecture Notes in Chemistry 22 - The Unitary Group,'* (Springer-Verlag, 1981), ed. J. Hinze.

[9] This number is given by the hardware performance monitor on the CRAY-YMP at SCRI, Tallahassee Fl., running the sample calculation described in Section 2.7 using the COLUMBUS program system [10].

[10] R. Shepard, I. Shavitt, R.M. Pitzer, D.C. Comeau, M. Pepper, H. Lischka, P.G. Szalay, R. Ahlrichs, F.B. Brown, and J.-G. Zhao, *Int. J. Quant. Chem.* **S22** (1988) 149.

[11] R.J. Harrison, J. Chem. Phys. (1990), submitted for publication.

[12] J.A. Pople and W.J. Hehre, J. Comp. Phys. **27** (1978) 161.

[13] V.R. SAUNDERS IN *'Methods in Computational Physics,'* NATO ASI, Bad Windshein, West Germany, (D. Reidel, Dordrecht, 1982).

[14] S. OBARA AND A. SAIKA, J. Chem. Phys. **89** (1989) 1540.

[15] L.E.M. MCMURCHIE AND E.R. DAVIDSON, *J. Comp. Phys.* **26** (1978) 218.

[16] M. DUPUIS, J. RYS AND H.F. KING, *J. Chem. Phys.* **65** (1976) 111.

[17] P.M.W. GILL, M. HEAD-GORDON AND J.A. POPLE, *Int. J. Quant. Chem.* **23** (1989) 269.

[18] J. ALMLÖF, J. FAEGRI, JR. AND K. KORSELL, *J. Comp. Chem.* **3** (1982) 385.

[19] C.C.J. ROOTHAAN, *Rev. Mod. Phys.* **23** (1951) 69.

[20] G.C. HALL, *Proc. Roy. Soc. Series A* **205** (1951) 541.

[21] S.T. ELBERT IN *'Numerical Algorithms in Chemistry: Algebraic Methods - LBL 8158,'* ed. C. Moler and I. Shavit (Lawrence Berkley Laboratory, University of California, Berkeley, 1978), 129.

[22] C.F. BENDER, *J. Comp. Phys.* **9** (1972) 547.

[23] V.R. SAUNDERS AND J.H. VAN LENTHE, *Mol. Phys.* **48** (1983) 923.

[24] T.H. DUNNING, *J. Chem. Phys.* **90** (1989) 1007.

[25] G. CORONGIU AND J.H. DETRICH, *IBM Technical Report* **KGN-1**, March 31, 1984.

[26] G. CORONGIU AND J.H. DETRICH, *IBM J. Res. Dev.* **29** (1985) 422.

[27] E. Clementi, G. Corongiu, J.H. Detrich, H. Khanmohammadbaigi, S. Chin, L. Domingo, A. Laakosonen and H.L. Nguyen, *IBM Technical Report* **KGN-2**, May 20, 1984.

[28] J.E. Watts, M. Dupuis and H.O. Villar, *IBM Technical Report* **KGN-78**, August 29, 1986.

[29] M.F. Guest, R.J. Harrison, J.H. van Lenthe and L.C.H van Corler. *Theor. Chim. Acta* **71** (1987) 117.

[30] M. Dupuis and J.D. Watts, *Theor. Chim. Acta* **71** (1987) 91.

[31] R. Ernenwein, M. Rohmer and M. Benard, *Comp. Phys. Commun.* **58** (1990) 305.

[32] M.E. Colvin, Ph.D. Thesis, '*Quantum Chemical Methods for Massively Parallel Computers,*' Department of Chemistry, University of California, 1986.

[33] N.S. Ostlund, '*Super-Computers in Chemistry,*' London, 1983.

[34] M.F. Guest, private communication, 1989.

[35] R.J. Harrison and R. A. Kendall, *Theor. Chim. Acta*, submitted for publication, 1990.

[36] P.R. Taylor and C.W. Bauschlicher, Jr., *Theor. Chim. Acta* **71** (1987) 105.

[37] J.S. Binkley, private communication, 1989.

[38] R.A. Whiteside, N.S. Ostlund and P.G. Hibbard, *IEEE Transactions on Computers* **C33** (1984) 409.

[39] P.J. Eberlein and H. Park, *J. Par. Dist. Comp.* **8** (1990) 358.

[40] K.A. GALLIVAN, R.J. PLEMMONS AND A.H. SAMEH, *SIAM Review* **32** (1990) 54.

[41] J.N. HURLEY, D.L. HUESTIS AND W.A. GODDARD, *J. Phys. Chem.* **92** (1988) 4880.

[42] C.W. BAUSCHLICHER, JR., *Theor. Chim. Acta* **76** (1989) 187.

[43] J.D. WATTS AMD M. DUPUIS, *IBM Technical Report* **KGN-100**, January 31, 1987.

[44] J.D. WATTS AMD M. DUPUIS, *J. Comp. Chem.* **9** (1988) 158.

[45] R.A. WHITESIDE, J.S. BINKLEY, M.E. COLVIN AND H.F. SCHAEFER III, *J. Chem. Phys.* **86** (1987) 2185.

[46] L.A. COVICK AND K.M. SANDO, *J. Comp. Chem.* (1990), in press.

[47] L.A. COVICK, AND K.M. SANDO, in preparation, 1990.

[48] A. EKANADHAM AND K. EKANADHAM, *J. Par. Dist. Comp.* **8** (1988) 460.

[49] I. FOSTER AND S. TAYLOR, *'Strand, New Concepts in Parallel Programming'* (Prentice Hall, New Jersey, 1990).

[50] N. CARRIERO AND D. GELERNTER, *ACM Computing Surveys*, November, 1989.

[51] J. BOYLE, R. BUTLER, T. DISZ, B. GLICKFELD, E. LUSK, R. OVERBEEK, J. PATTERSON AND R. STEVENS, *'Portable Programs for Parallel Processors,'* (Holt, Rinehart and Winston, Inc., 1987).

[52] R. PITZER, *J. Chem. Phys.* **58** (1973) 3111.

[53] C.W. BAUSCHLICHER, JR. AND P.R. TAYLOR, *J. Chem. Phys.* **85** (1986) 2779.

Table 1: Timings for parallel AO integral evaluation on a SUN SparcStation Ethernet LAN. The system is Si_4H_9CaF in a DZ basis (101 functions) with no symmetry. The CPU per process indicates the range of cpu time processes spent evaluating two-electron integrals, and excludes time spent evaluating one-electron integrals. The number in parentheses is the sum of these times. The efficiency is computed from the wall time for the entire job.

No. of Processors	CPU seconds per process	Wall time (seconds)	Efficiency (percent)
1	10736	14683	100
	(10736)		
2	5294-5671	7392	99
	(10965)		
3	3713-3500	4859	101
	(10920)		
4	2836-2545	3782	97
	(10950)		
5	2246-2380	3258	90
	(11510)		

Table 2: Timings for parallel AO integral evaluation on a SUN SparcStation Ethernet LAN. The system is the carbon dimer in an augmented cc-pvqz basis basis [23] (160 functions) with D_{2h} symmetry. The CPU per process indicates the range of cpu time processes spent evaluating two-electron integrals, and excludes time spent evaluating one-electron integrals. The number in parentheses is the sum of these times. The efficiency is computed from the wall time for the entire job.

No. of Processors	CPU seconds per process	Wall time (seconds)	Efficiency (percent)
1	24450 (24450)	24764	100
2	11889-12152 (24041)	12472	99
3	8606-7923 (24872)	8753	94
4	5964-6163 (24257)	6390	97
5	4463-5373 (24684)	5500	90

Table 3: Timings for a tiny selected CI+PT calculation. Times are wall-clock seconds for the entire job. The number of processors of each type (SUN-4/110, Alliant FX/8, Ardent Titan) is indicated, with one process per processor. The actual computation is on the Be atom in a 9s5p basis set using D_{2h} symmetry, the reference being all singles and doubles from the Hartree-Fock reference (285 CSF).

Total no. of	Process Placement			Wall Time	Efficiency
processes	SUN-4/110	Titan	Alliant	(seconds)	(percent)
1	1	0	0	104	100
2	2	0	0	57	91
1	0	1	0	105	100
2	0	2	0	61	86
5	0	5	0	46	46
1	0	0	1	144	100
2	0	0	2	82	88
4	0	0	4	54	67
7	0	0	7	48	43
3	1	1	1	53	72[1]
6	2	2	2	50	38[1]

[1]The 100% efficient parallel runtime (t) for n_i processors which execute a single process job in time t_i, is taken as $\frac{1}{t} = \sum_i n_i/t_i$.

Table 4: Timings for a small selected CI+PT calculation. Times are wall-clock seconds for the entire job. The number of processors of each type (SUN-4/110, Alliant FX/8, Ardent Titan) is indicated, with one process per processor. The actual computation is on water in a DZP basis set using C_{2v} symmetry, a selection threshold of $0.00003125 E_h$, yielding 3261 reference functions. The resulting energy is $-76.2563 E_h$, to be compared with the exact full-CI eigenvalue of $-76.2566 E_h$ [52].

Total no. of	Process Placement			Wall Time	Efficiency
processes	SUN-4/110	Titan	Alliant	(seconds)	(percent)
1	0	0	1	11780	100
2	0	0	2	5958	96
4	0	0	4	3064	96
6	0	0	6	2130	92
7	0	0	7	1865	90
8	0	0	8	1659	89
12	1	5	6	1076	N/A