Title
VECTOR AND PARALLEL COMPUTERS FOR QUANTUM MONTE CARLO COMPUTATIONS

Author
Reynolds, P.J.

Publication Date
1985-08-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Materials & Molecular Research Division

Presented at the Conference on Supercomputer Simulations in Chemistry, Montreal, Quebec, Canada, August 25-27, 1985

VECTOR AND PARALLEL COMPUTERS FOR QUANTUM MONTE CARLO COMPUTATIONS

P.J. Reynolds, S. Alexander, D. Logan, and W.A. Lester, Jr.

August 1985

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

Vector and Parallel Computers for Quantum Monte Carlo Computations*

P. J. Reynolds[a], S. Alexander[a] †, D. Logan[b] ‡, and W. A. Lester, Jr.[a] §

[a] Materials and Molecular Research Division and
[b] Advanced Computer Architecture Laboratory
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Abstract

Monte Carlo simulations are inherently compute-bound. Although short computations may provide order-of-magnitude estimates, long CPU times are generally required to achieve the accuracy needed for reliable comparison of Monte Carlo results with experiment or theory. The advent of supercomputers, which have made possible significantly increased computer speeds for those applications which are amenable to vector or parallel processing, thus offers promise for Monte Carlo applications. In fact, Monte Carlo codes are often highly parallel, and offer multiple avenues for both parallelisation and vectorization.

We explore the gains to be obtained with supercomputers for the quantum Monte Carlo (QMC) method. The QMC algorithm treated here is used in quantum mechanical molecular calculations, to obtain solutions to the Schrödinger equation. This approach has recently been shown to achieve high accuracy in electronic structure computations. QMC is here demonstrated to fully take advantage of parallel and vector processor systems. Levels of parallelism are discussed, and an overview of parallel computer architectures, as well as present vector supercomputers is given. We also discuss how one adapts QMC to these machines. Performance ratios (versus scalar operation) for a number of supercomputer systems are given.

---

* This work was supported by the director, Office of Energy Research, Office of Basic Energy Sciences, Chemical Sciences Division of the U. S. Department of Energy under contract number DE-AC03-76SF00098.

† Present address: Quantum Theory Project, University of Florida, Gainesville, FL 32611

‡ Permanent address: IBM Research Laboratory, Kingston, NY 12401

§ Also, Department of Chemistry, University of California, Berkeley, CA 94720

I. Introduction

Many-body problems in physics are often treated by a Monte Carlo approach [1]. The Monte Carlo method is statistical in nature; based on the generation of "random" numbers or "coin tosses," it derives its name from a city famous for the random numbers embodied in its games of chance. Easy as it is to imagine using the Monte Carlo method for treating inherently statistical models, or even for numerical integration [2], it is less obvious how to solve many-body problems. Nevertheless, many such problems are readily treated by Monte Carlo. Of particular interest to chemistry are the quantum mechanical Monte Carlo methods [3]. These have recently been applied sucessfully to a number of molecular problems [4-9], where they have achieved very high accuracy compared to experiment and exact results (where available). In most cases, 90-100% of the correlation energy has been obtained. Such quantum Monte Carlo (QMC) approaches stochastically *solve* the Schrödinger equation. Thus QMC provides an alternative to the conventional techniques of quantum chemistry.

Monte Carlo algorithms are almost entirely compute-bound, doing almost no I/O and requiring minimal memory. Since Monte Carlo is a statistical procedure, high precision (i.e., knowledge of many decimal places) can require long computer runs. This is because statistical uncertainty decreases slowly with computational time, going as $c\,(time)^{-\frac{1}{2}}$, where c is a constant. Because high precision is required in chemistry, where most properties are obtained as differences of almost equal large numbers, long computation times are often necessary.

One direction in addressing the problem of high precision is strictly algorithmic. An algorithm which has a significantly smaller value for the constant c will run faster by the ratio of these constants. Importance sampling [2,3] falls into this category of algorithmic approach. Other algorithmic approaches, like differential Monte Carlo methods [10], increase precision by computing energy differences directly. A

complementary direction in obtaining higher precision is machine oriented. In Monte Carlo applications, increases in computational speed translate into increases in precision. Vector and parallel processors promise vast enhancements in speed for those codes able to take advantage of these architectures. Most Monte Carlo codes, and the molecular QMC code in particular, can use these architectures extremely efficiently.

In Section II we highlight some features of QMC theory and the corresponding algorithm, exploring levels of parallelism. In Sect. III we present an overview of *vector* processors, vectorizing QMC for these machines, and the enhancements in speed thus obtained. Sect. IV gives a discussion of *parallel* processing, including the various types of architectures which are categorized under this heading. Further, we discuss the gains that QMC can obtain through the use of parallel machines in terms of efficiency in using all processors. In conclusion, Sect. V gives a brief comparison of the quantum Monte Carlo speed enhancements achieved on supercomputers with those achieved by *ab initio* methods.

II. Quantum Monte Carlo

For the purpose of this discussion, our goal will be the solution of the time-independent Schrödinger equation $H\Psi=E\Psi$ for the energy E. Expectation values, $<A>=<\Psi \mid A \mid \Psi>$, can also be obtained, however these details [11] obscure the main features of the approach, and so here we focus on E. It is readily apparent that the ground state of the time-independent Schrödinger equation is the steady-state solution to the equation [5,6]

$$\frac{\partial \Psi}{\partial t}=D\,\nabla^2\Psi + [E-V(\underline{R})]\Psi(\underline{R},t)\,. \qquad (1)$$

Here $\underline{R}$ is the vector of coordinates of all N particles in the system. If the particles being treated quantum mechanically are all electrons, ∇^2 is simply the 3-N dimensional

Laplacian and $D = \hbar^2/2m_e$. Note that Eq. 1 is simply a diffusion equation combined with a first-order rate process. The function $\Psi(\underline{R},t)$ has the meaning of the density of diffusing particles, which may increase or decrease locally according to the rate term $[E-V(\underline{R})]\Psi(\underline{R})$. Such a diffusion process is readily simulated on a computer, and such simulations have been performed for simple molecules as long as a decade ago [12].

A key issue in treating molecular systems is the proper handling of the Fermi nature of electrons. The requirement that the wave function Ψ be antisymmetric with respect to particle exchange leads to a wave function which, for more than two electrons, must have negative as well as positive values. In this situation, Eq. 1 may no longer be simulated as a diffusion process, since the density of diffusers $\Psi(\underline{R},t)$ would no longer be everywhere positive. A number of methods have been proposed to deal with fermions [13,14]; the most stable algorithmically, as well as precise statistically, entails a small variational approximation known as the fixed-node approximation [13]. The antisymmetry constraint on $\Psi(\underline{R})$ is built in by specifying in advance the nodes of the wave function to be those of a trial wave function $\Psi_T(\underline{R})$. A random walk simulation is performed separately in each volume element bounded by these nodes. Since the sign of Ψ does not change within a volume element, the problem with Ψ being a density is removed. (Excited-state Fermi calculations may be done similarly, using such built-in nodal constraints [15].) The function $\Psi_T(\underline{R})$ can further serve as a guiding function for importance sampling, during the Monte Carlo random walk. Such a guided walk preferentially samples regions of configuration space where the true wave function is expected to be large, and avoids divergent rate coefficients, such as that which occurs in Eq. 1 when an electron approaches either another electron or a nucleus.

The result of introducing importance sampling and the fixed-node approximation into Eq. 1, is to leave it as essentially a diffusion equation with branching -- though now

with an overall drift, and a modified branching term. The resulting equation is solved stochastically by allowing a set of diffusers ("random walkers") to evolve in time until a steady-state asymptotic distribution, is obtained [5]. Properties such as the energy are measured as the walkers proceed in this asymptotic distribution, and are thus averages over that distribution. The time evolution is achieved by using a Green function $G(\underline{R} \rightarrow \underline{R}', \tau)$ which evolves a walker at time t at coordinate $\underline{R}$ to time $t+\tau$ at coordinate $\underline{R}'$. The algorithm which performs this time evolution by Monte Carlo is quite simple. It is summarized here.

(1) Nested loops over initial conditions. In most cases, a number of different initial conditions are of interest. For example, in calculating a potential-energy surface, various separate calculations at different nuclear geometries must be performed. In other calculations, a number of different trial functions may be used, for example to ascertain basis-set dependence. In still other instances a number of calculations with different time-steps τ are required for extrapolation to $\tau \rightarrow 0$, if the short-time approximation [5,6] is used. One may also want to perform a set of runs differing only in the initial "seed" for the random number generator. Such runs would not suffer from the serial correlation present in the block averages (see below). It is furthermore conceivable (and common) to need to loop over multiple initial conditions. For example, one may need to loop over geometries, at each geometry perform a loop over τ, and perhaps, at each τ perform a loop over random seeds. Each of these loops is entirely independent of one another.

For a given set of initial conditions, one generates an ensemble of N_c (typically 100-500) spatial "configurations" of the N-electron system. These coordinates may be chosen randomly, or for greater efficiency in reaching the asymptotic distribution, they may be drawn from the distribution $|\Psi_T(\underline{R})|^2$.

(2) Loop over blocks. A block is an almost statistically independent

"sampling unit." Each block is a complete Monte Carlo "run," which provides estimates of the properties being sampled. Generally, the total microscopic sampling time (or "target time") in a block is taken as approximately one atomic unit or longer, in order to minimize the statistical (serial) correlation between blocks. This target time defines the length of a block. Calculation of a block entails:

(3) Loop over configurations in the ensemble. For a *particular* configuration, one "measures" the properties of interest, such as the energy. (When using a trial function, the energy measured is the local energy. In the case of Eq. 1, the potential energy is measured.) Perform the random walk by:

(4) Loop over the electrons. Each electron will diffuse and drift as prescribed by the Green function. One must check if a node has been crossed in this process. In the fixed-node approximation, if the answer is "yes," eliminate the configuration. If no,

Continue the loop over electrons.

After all electrons have been moved, one advances the time in the current configuration by τ. The branching factor, or multiplicity M, which is also given by the Green function, is computed. This factor comes from the rate part of the differential equation (see e.g. Eq. 1). M copies of the current configuration are placed in the ensemble in place of the starting configuration. Averages for this configuration

are weighted by M.

Continue the loop over configurations.

At this point, all surviving configurations in the ensemble have reached the target time. The current block is finished. Store the current averages. "Renormalize" the ensemble back to N_c to avoid the otherwise inevitable overflow, or total death, of configurations in the ensemble.[3]

Continue the loop over blocks.

The pre-determined number of blocks have now been computed. Averages and standard deviations over the blocks are computed. This provides a Monte Carlo estimate of the mean and standard errors of the quantities of interest for a particular initial condition.

Continue the loops over initial conditions.

In this outline, we have not included many of the details of the calculation, in order to focus more on the levels of parallelism and nesting. We *have* mentioned the key loops--many of which are totally independent of one another. This allows for a variety of options in creating efficient vector and parallel algorithms. The nominal communication overhead is particularly attractive, as it is beneficial in virtually every parallel architecture. We note, however, that the choice of loops vectorized can have an appreciable impact on the amount of memory required. Replicating the whole program over initial conditions is only practical if the original code uses very little memory--as is the case for QMC.

III. Vector processors

Scalar computers perform sequential arithmetic operations on individual data elements. These are single-instruction, single-data (SISD) architecture machines. In contrast, vector processors are designed to perform identical arithmetic operations simul-

taneously on multiple data elements (single-instruction, multiple-data or SIMD). Thus, in order that vectorization can be performed, there needs to be one operation to be performed on many data elements at a given moment. (However, only certain operations on these elements are allowed.) Generally the compiler, together with some user directives, is responsible for finding such operations. What the compiler can recognize, even with user directives, is critically dependent on a program's overall structure. Thus structure as much as such conventional factors as cycle time, compiler efficiency, and memory access time, has a major effect on speed. A useful measure of the adaptability of a specific program to a vector machine is the ratio of its scalar run time to its vector run time on the same machine. For inherently scalar programs this number will be close to unity. Programs which make full use of a vector processor's capabilities will have much larger ratios. To examine the adaptability of QMC to vector processors, we performed test runs on a two-pipe and a four-pipe CDC Cyber 205, and on a Cray 1S and one processor of a Cray XMP.

The CDC Cyber 205 [16], consists of a fast scalar processor in addition to a memory-to-memory vector processor. Vector arithmetic instructions are executed in two phases. General-purpose functional units (pipes) are initially filled or emptied during the startup phase. The time consumed during this period is independent of the vector length, and typically takes on the order of 50 machine cycles. During the stream phase, CPU time consumed is directly proportional to the vector length, and inversely proportional to the number of pipes. On a two-pipe machine, for example, a vector addition uses the first pipe to add the odd pairs of operands, while the second pipe simultaneously adds the even pairs. This sort of operation yields two results per cycle. Similarly a four-pipe machine produces four results every cycle. Up to a maximum of four pipes are available on the Cyber 205. It should be noted, however, that not all vector opera-

tions benefit from increasing the number of pipes. Furthermore, for a fixed number of pipes, certain operations are performed faster than others. Thus the ultimate speed attainable on the Cyber 205 is dependent not only on the degree to which the code can be vectorized, but also on the type of vector operations being performed. Further, although floating-point operations are normally done in full precision (64 bits), a facility has been provided for the user to code in half-precision (32 bits). In most cases this doubles both the speed and the amount of memory available [17]. Another feature of the Cyber 205 is the large, diverse instruction set available to the FORTRAN user. This enables one to convert many standard FORTRAN constructs directly into vector instructions. Thus vectorization of an algorithm does not depend strictly on the cleverness of the compiler.

The Crays [18], like the Cyber 205, combine fast scalar capability with vector processing. Unlike the Cyber 205, however, Crays are not memory-to-memory machines. Instead, vectors are loaded from memory into one of 8 vector registers, each holding 64 words of 64 bits each. From these registers the numbers are sent to one of 12 specialized functional units where arithmetic operations are performed. Operations involving seperate functional units can proceed concurently. All result vectors are returned to the vector registers. Since each operand does not have to be fetched or each result stored in memory, operations are performed much faster. The vector registers act as fast cache memory. As a result Cray vector instructions are characterized by relatively short startup times, ranging from 2 to 14 clock cycles. Clock cycles range from 12.5 nanoseconds on the Cray 1S to 9.5 ns on the Cray XMP -- in constrast to 20 ns for the Cyber 205. On the other hand, Cray machines have a relatively small number of instructions available to the FORTRAN user, which can hinder vectorization of some codes.

Direct quantitative comparison of the Cyber 205 and the Crays is difficult because their different capabilities give rise to different optimal coding techniques. Nevertheless, some general conclusions may be drawn. In applications involving short vectors, the Crays seem to be superior due to their shorter startup times. Their faster clock cycles will also give them an advantage in many applications. However, for long loops, and in cases where explicit vector instructions are necessary, the Cyber 205 (especially with 4 pipes) may be more desirable, due to its higher processing rate per machine cycle, and its large, diverse set of vector FORTRAN instructions.

The QMC algorithm, as described in Sect. II, is well suited to scalar machines. However, this structure prevents it from being vectorized efficiently. Typically only the inner loops of a program can be vectorized. The longer a loop the more efficient the process [19]. The innermost loops, however, are over the electrons and over the basis functions. The number of electrons, and even the number of basis functions, is relatively small for the systems currently being considered. For example, vectorizing these loops for CH_2 [20] yields a scalar/vector ratio very close to unity on both the Cray 1 and the Cyber 205. A similar result is obtained for the carbon atom [21]. This algorithm is clearly not taking advantage of the vector architecture. Even writing explicit vector code (for the Cyber) [20] and using it only for vectors longer than an optimized length $C^* \approx 16$ (optimized to minimize the effect of the vector start-up time) leads to a scalar/vector ratio only in the range of 1-1.5, depending on the number of basis functions. Much longer vectors are clearly needed for these machines to show their abilities.

A more appropriate arrangement of the code -- that will allow the compiler to create a long vector -- is to make the loop over configurations innermost. This involves storing a linear array the size of the ensemble for every quantity in the original loop over configurations. Since each configuration is independent of the others, and since the

number of configurations is usually quite large, this restructured algorithm should be much more efficient. Certain operations in the inner loop pose something of a problem, however, since they are performed on some configurations but not others. These can nevertheless be vectorized by using special mask operations available on all the machines we considered. The restructured algorithm does, however, require considerably more memory than the original form. Nevertheless, the minimal memory requirements of the scalar algorithm are such that the vector code still requires only a small memory allocation.

We note that even the scalar VAX time obtained with the restructured algorithm shows a speed-up in execution of about 40%. This type of tradeoff between speed and memory is common. When automatic vectorization is invoked on the Crays and Cybers, CPU time drops by a factor of roughly three. The current generation of compilers, however, have only a limited capability to recognize vectorizable code. To achieve faster execution it was necessary to explicitly hand-vectorize portions of the program. In Tables 1 and 2 we present the results achieved. Scalar/vector ratios range from 5-6 for a vector length of 100, and from 6-10 for a vector length of 500. (In actual QMC calculations the ensemble size is generally in the range of 100 to 500.) Thus overall, for a vector length of 500, we achieve a factor of 117-186 over a VAX 11/780 running the same code, and a factor of 192-304 over the same VAX running the original algorithm. It is also important to note that in comparing speeds with the VAX we are comparing single precision on the VAX (32-bits--this is all that QMC requires in most cases) with single precision on the supercomputers (64 bits). For comparisons with equal numbers of significant figures, one must use single precision on the VAX and half precision on the supercomputers (where available), or double precision on the VAX and single precision on the supercomputers. In such a comparison (for a vector length of 500) we expect to

achieve a speed-up over the VAX (for the same code) of roughly 370 on the Cray XMP and about the same on the four-pipe Cyber 205. This is a significant increase, especially when compared with the naive vectorization [20] of the scalar algorithm, which only led to factors of 20 to 30. Expressed in millions of floating-point operations per second (MFLOPS), we obtain an average rate of roughly 70 MFLOPS on the Cray XMP and on the four-pipe Cyber 205 at a vector length of 500. This is to be compared to the 14 Livermore kernels, which range from 3-150 MFLOPS on the Cray XMP, and average either 50 or 58 MFLOPS depending on the use of compiler directives [22a]. The harmonic mean, which is a better indicator of the actual speed of a code running sections going at different rates, gives an average rate of roughly 14 MFLOPS for the Livermore kernels on the Cray XMP [22b]. On the other hand, in half-precision we would expect our code to achieve an average rate of about 140 MFLOPS. We note especially that these rates are for the code *overall,* not just for selected parts of it.

IV. Parallel Processing

A more ambitious approach than the SIMD vector machines are the fully parallel MIMD (multiple-instruction, multiple-data) structures, in which parallelism is achieved at the *processor* level, rather than with the functional units. The two approaches, however, should not be viewed as mutually exclusive. Ideally, one can envision a system that encompasses both strategies, i.e., a multiprocessor system whose component processors are capable of vector processing. Such an approach allows for more rapidly solving those problems that are vector decomposable, as well as solving a multitude of problems that are not [23]. QMC (and other Monte Carlo) can be decomposed simultaneously into vector and parallel parts, gaining from both types of architecture.

A number of new computational issues arise when approaching a parallel processing system. The first, of course, is in understanding the level of parallelism that a problem

manifests. Such parallalism may exist, for example, at the "fine-grained" instruction level. Pipelining techniques of the vector supercomputers currently avail themselves of this form of parallelism. More ambitious approaches have been concerned with the construction of multiple-instruction pipes [24] and with data-flow architectures [25]. At the opposite extreme, the highest level of parallelism may be the segmentation of a problem into a group of concurrent co-operative subtasks, or even replication of the entire program with, for example, differing initial conditions, constraints, etc. (see Sect. II).

Closely related to the understanding of a program's parallelism is the question of how best to express this in a programming language. Is it adequate to modify an existing language with appropriate constructs, as done for vectorization, or must a new language be defined? Further, to what relative degrees shall the programmer or the compiler be responsible for the program decomposition? To a first approximation, it is generally conceded that the finer the grain size, the more emphasis must be placed upon the "intelligence" of the compiler. The higher levels will require greater participation of the programmer with perhaps interactive compilers.

Even with some understanding of the level of parallelism that a problem exhibits, and of how to map it onto a system of co-operative processors, there remains the critical issue of how best to implement processor coordination. This is related to the general issue of communication. Two broad approaches have been most extensively analysed in this regard: tightly- and loosely-coupled systems. Tightly-coupled systems may be thought of as a collection of processors that share a global memory. Important for such systems is the selection of a processor/memory interconnection network [26]. This network must ensure that the potentially very high interprocessor communication bandwidth is not degraded by memory contention conflicts and slow arbitration schemes. Further, in the event that the processors retain an additional private memory or data

cache, it is vitally important that local data be correct in a global sense, i.e., data updates or modifications must be distributed across private memory boundaries. This latter criterion is the essence of what is known as the cache coherency problem [27]. The loosely-coupled alternative approach encompasses multiprocessor systems that operate as high-speed local-area networks. Here the problem of resource contention and validity is replaced by the concern with how interprocessor communication may be most effectively implemented, and the degree of system-wide connectivity that the applications merit. In the event that complete pairwise connectivity is required, bus structures may be employed. Higher communication band-width, however, requires more elaborate and costly interconnection networks [26]. Alternatively, in some applications static geometries of processors with solely nearest-neighbor communication abilities are sufficient. These generally are special-purpose machines with only a few applications. A number of such systems have been designed with topologies such as grids, rings, trees, pyramids, hypercubes, and other exotic structures [26,28]. In examples such as these, the architecture is designed to embody as closly as possible the "model of computation" required by the application. Every problem has some "natural connectivity" [28]. Having established the connection scheme, there still remains the specification of communication protocols and communication method (e.g., packet switching, memory circuit switching, etc.).

However the issue of processor coordination is resolved--whether in favor of loosely- or tightly-coupled architectural approaches--there still remain other issues. For example, should an application require intensive data throughput ($>$10 Mbyte/sec), it may be necessary to design specialized input and output processors, an additional interconnection network to distribute data, and sophisticated mass storage subsystems. Other system-level issues to be dealt with are job dispatching, fault tolerance, the ability to

control synchronization and data broadcasting among processors, and of vital importance, the ability to monitor system performance while experimenting with problem decomposition.

At present there exists no consensus as to what approach is best. While many systems have been suggested, relatively few have actually been constructed and tested. Moreover, those that have, of necessity have been prototypes with small numbers of processors ($\approx$ 2-64) and usually with limited memory and I/O capability. Thus the benchmarking of real, large-scale problems has consisted primarily of extrapolation of the results obtained on these prototypes, into the region of the large numbers of processors ($\approx$ 100-1000) envisioned for future systems. These considerations led to the design and construction of a parallel processing system called MIDAS [29] at the Advanced Computer Architecture Laboratory at the Lawrence Berkeley Laboratory. MIDAS (Modular Interactive Data Analysis Systems) has combined many of the advantages of both loosely- and tightly-coupled architectures (high communication speeds without resource contention), and dealt explicitly with I/O intensive problems.

Since Monte Carlo problems are so ideally suited to multiprocessor systems, we have adapted QMC to MIDAS (and MIDAS to QMC) [30]. The most intuitive and usually most efficient means of distributing a Monte Carlo computation is such that each processor is responsible for performing an independent statistical sample. An ensemble average replaces a time average in those Monte Carlo applications where equilibrium time averaging is performed. Actually, one has an ensemble average of time-averaged quantities. Increasing the number of processors substitutes more elements in the ensemble for parts of the time average. This, however, also raises the overhead of equilibrating all members of the ensemble. Some Monte Carlo applications involve only ensemble averaging, and these are most readily modified for parallel execution. The decomposition

of QMC chosen for MIDAS was to give each procesor initially an equal fraction of the total ensemble. This choice, rather than e.g. breaking up the calculation over blocks or initial conditions, was dictated by the limited memory size of the individual processors (128 Kwords).

In Monte Carlo, the necessity of inter-processor communication is primarily dependent on whether the sampling conditions or rules change as a function of previous sample estimates. In the event that such conditions remain constant, communication may be unnecessary (other than that required for averaging the final estimates of each processor). If, on the other hand, the set of estimates from all processors needs to be assessed for the purpose of defining a new sampling condition, periodic communication is required. Such a synchronization step represents a "critical section" of the problem. The computation may be thought of as a "fork and join" process, wherein processors fork to arrive at their estimates, followed by a join operation to define the new sampling condition. The join represents the critical section that must be performed prior to another fork. This type of critical section was explicitly realized in the QMC calculation: Each processor calculated an estimated energy by allowing its configurations to perform random walks for a fixed number of time steps. When the last processor completed this operation, all local estimates of the energy were combined to update the trial energy for the next sample period.

The MIDAS structure can be configured in a number of ways. The structure chosen to implement QMC was that of a master/slave topology. In this configuration, a single processor acts as the master and performs the join operation of updating the trial energy. Thus it was responsible for polling the slaves to ascertain whether they had completed their samples. In the event that all were completed, it read the current energy value, recalculated the trial energy, wrote this value into each slave processor's

memory, and initiated (forked) the next sample period. At a lower level, the master was also responsible for detecting abnormal conditions or failures in each of the slave processors (underflow, overflow, memory parity errors, etc.). In the event that such conditions were uncorrectable, the master deallocated the processor in question, while increasing the sub-ensemble populations in the remaining processors such that the total ensemble remained constant. If any processor's population of configurations died entirely during a sample period (as happens more frequently with larger time steps) the master was responsible for downloading a new set of configurations to that processor prior to initiating a new sample period. This process of augmenting a processor's population was made by randomly selecting surviving configurations from neighbor processors.

Modifications to the original serial QMC code were fairly minor [30]. A subroutine was added that performed all of the master functions described above. It is called in the master once all initial conditions have been set, and the sampling portion of the code downloaded and initiated in each slave processor. This routine, consisting of several hundred lines of FORTRAN, executes system routines [31] which read and write selected variables within slave processors' memories, which poll processor status conditions, and which start execution at selected program addresses. Modification of the QMC code that was executed in each slave was minimal. It consisted of setting software flags for various conditions at the end of a sample period, and thereafter calling a system routine that suspended execution at a particular instruction address. This latter address and condition was recognized by the master as the termination of a normal sample period.

We now address how well QMC performs on a parallel system. Since the efficiency of a multiprocessor calculation may be defined as the fraction of time that an average processor is employed in performing useful work, 100% efficiency corresponds to a P-fold increase in speed over a uniprocessor, for a P-processor machine. Because of the random

nature of the QMC birth/death process, the time for a given processor to complete (before the next "join") is itself randomly distributed. Hence, in this implementation, a processor upon completion of its calculation, is forced to remain idle until the last processor has finished. Only at that point does the master compute the new trial energy and reinitiate the slave processors. Thus the efficiency can be approximated as t_{ave}/t_{max}, where t_{ave} is the average time at which the processors finish and t_{max} is the average time at which the last processor finishes. With MIDAS configured with 8 slave processors, calculations were performed for the saddle-point energy of H_3, and the ground-state energies of N and of N_2. These calculations ran at approximately 95%, 85%, and 80% efficiencies respectively. Decreasing the number of processors increased the efficiencies to a small extent, but, of course, at the expense of decreasing the overall computational rate. The decrease in efficiency with the number of processors is attributable to t_{max} increasing with P, as one samples further under the tail of the distribution of finishing times. The decrease in efficiency with the larger molecules may be attributed to sampling from a broader distribution.

Although the actual calculations were performed as described above, the parallel algorithm can be readily modified to improve overall performance. The master can be programmed to perform dynamic load balancing as the sampling process proceeds. Any processor completing its work can be given additional configurations, extracted from processors which still have unfinished ones waiting to run. It should also be noted that the need for even the minimal interprocessor communication, and its resultant inefficiency, could have been entirely eliminated had each slave processor had sufficient, memory to hold the entire ensemble. In this case the loop over blocks or any (or all) of the loops over initial conditions (see Sect. II) could have been decomposed over the processors. Thus each processor could have run at 100% efficiency. This demonstrates clearly how

the actual strategy for parallelising depends on external factors, such as memory constaints.

The number of available processors also dictates the strategy for parallelising an application. Consider the availability of an unlimited number of processors. Then an ideal parallelisation of QMC would follow the nested nature of the algorithm. At the top level, independent processors would run differing sets of initial conditions--e.g. nuclear geometries. Each such processor, however, would itself be a parallel processor, running multiple (identical) codes, differing e.g. in the time-step size. Each of these processors would be a multi-processor also. These next-lower-level processors could run statistically independent samples by using different random number generators, or (if this is deemed unnecessary) by using differing seeds for the same generator. These processors could be further subdivided, with separate processors handling parts of the ensemble -- as was actually implemented on MIDAS and described above. All the above steps can be run at virtually 100% efficiency, leading to increases in execution speed of P, the number of processors. In a truly massively parallel architecture, if there are more processors available than can be used in this description, one can further parallelise the random-walk algorithm, for example, parallelising the computation of the Coulomb potential. At this stage, efficiency would begin to drop, although speed would continue to increase, but no longer as rapidly. At a certain point saturation will set in, and speed will no longer rise even linearly with the number of processors. For QMC however, this stage apparently will not be reached until P is quite large indeed.

V. Comparison with conventional *ab initio* methods.

As we have seen, Monte Carlo is a computationally intensive method, but one that requires relatively little memory. The independence of the configurations, the near independence of the blocks, and the possibility of parallelising over initial conditions,

makes QMC particularly well suited to vector and parallel machines. Furthermore, the relatively small size of most Monte Carlo codes (roughly 1500 lines of FORTRAN in the present case) allows the user to easily optimize the algorithm to a specific machine. In this section we discuss whether these factors allow QMC to make more efficient use of current supercomputers and upcoming parallel computers, than more traditional *ab initio* methods (e.g., Hartree-Fock, multiconfiguration Hartree-Fock, configuration interaction) -- as the latter also benefit from the new computer architectures.

Scalar/vector ratios of from 10 to 50 have been quoted by Rappe [32] for selected portions of SCF and MCSCF codes on a two-pipe Cyber 205. The corresponding rates are from 26-100 MFLOPS. (This implies that at least in part his high ratios result from a scalar execution rate of only approximately 2 MFLOPS, in contrast to the more usual scalar rate of 5 MFLOPS.) It should also be noted that only 14% of his CPU time is spent in the sections going at the highest rates [32]. In programs containing a mixture of vectorizable and nonvectorizable code, the nonvectorizable part dominates the calculation [33]. The question of how much vectorization helps his SCF codes overall is not addressed. Sanders and Guest [34] discuss rates for somewhat larger program sections than Rappe. Their rates on a Cray 1S range from 10 MFLOPS for the Hartree-Fock section to 120 MFLOPS for the CI part. However the *overall* benefit remains an open question here too. (This question has, however, been addressed recently for *ab initio* calculations performed on an FPS 164 array processor [35] where relative to a VAX an overall enhancement of 10-12 was achieved.)

For QMC, virtually the *entire* code vectorizes. In fact, using a formula from Ahlrichs *et al* [36] we estimate that over 90% of the code must vectorize to achieve the rates attained. Further, QMC can be decomposed in a number of ways into virtually identical, non-communicating parts suitable for parallel processing. Although *ab initio*

codes can be decomposed over some initial conditions as well, the increase in memory required for such program replication makes this degree of parallelism impractical in most cases. Analysis of recent work in optimizing *ab initio* codes for parallel processors shows efficiencies ranging from 50-95%, depending on the type of code, and the number of processors [37]. We note, however, that these efficiencies are generally *not* linear in P, and begin to saturate (i.e. deviate from linearity) for P quite small (on the order of 4-10 processors). Thus, overall it appears that QMC can more easily and efficiently take advantage of the new computer architectures than conventional *ab initio* approaches.

Acknowledgements. We thank the Control Data Corporation and Cray Research Inc. for grants of computer time on their machines, and for assistance in carrying out parts of these calculations. Helpful comments on the manuscript by R. N. Barnett and B. L. Hammond are also gratefully acknowledged.

References.

[1] See, e.g., *Monte Carlo Methods in Statistical Physics,* K. Binder, ed. (Springer-Verlag, Berlin, 1979).

[2] J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods,* (Chapman and Hall, London, 1964).

[3] M. H. Kalos, Phys. Rev. 128, 1791 (1962); J. Comp. Phys. 1, 257 (1967); M. H. Kalos, D. Levesque, and L. Verlet, Phys. Rev. A 9, 2178 (1974); D. M. Ceperley and M. H. Kalos, in Ref. [1]; D. M. Ceperley, J. Comp. Phys. 51, 404 (1983).

[4] J. B. Anderson, J. Chem. Phys. 73, 3897 (1980); F. Mentch and J. B. Anderson, J. Chem. Phys. 74, 6307 (1981).

[5] P. J. Reynolds, D. M. Ceperley, B. J. Alder, and W. A. Lester, Jr., J. Chem. Phys. 77, 5593 (1982).

[6] J. W. Moskowitz, K. E. Schmidt, M. A. Lee, and M. H. Kalos, J. Chem. Phys. 77, 349 (1982).

[7] P. J. Reynolds, R. N. Barnett, and W. A. Lester, Jr., Int. J. Quant. Chem. Symp. 18, 709 (1984); F. Mentch and J. Anderson, J. Chem. Phys. 80, 2675 (1984); R. N. Barnett, P. J. Reynolds, and W. A. Lester, Jr., J. Chem. Phys., 82, 2700 (1985).

[8] P. J. Reynolds, M. Dupuis, and W. A. Lester, Jr., J. Chem. Phys. 82, 1983 (1985).

[9] R. N. Barnett, P. J. Reynolds, and W. A. Lester, Jr., "Electron Affinity of Fluorine: Quantum Monte Carlo Study " *in preparation.*

[10] B. Holmer and D. M. Ceperley, *private communication* ; B. Wells, P. J. Reynolds, and W. A. Lester, Jr., *unpublished* ; B. Hammond, P. J. Reynolds, and W. A. Lester, Jr., *unpublished* ; B. H. Wells, Chem. Phys. Lett. 115, 89 (1985).

[11] M. H. Kalos, Phys. Rev. A 2, 250 (1970); R. N. Barnett, P. J. Reynolds, and W. A. Lester, Jr., "Molecular Properties by Quantum Monte Carlo" *in preparation.*

[12] J. B. Anderson, J. Chem. Phys. 63, 1499 (1975); 65, 4121 (1976).

[13] D. M. Ceperley and B. J. Alder, Phys. Rev. Lett. 45, 566 (1980); D. M. Ceperley in *Recent Progress in Many-Body Theories* , edited by J. G. Zabolitzky, M. de Llano, M. Fortes, and J. W. Clark (Springer- Verlag, Berlin, 1981).

[14] J. Carlson and M. H. Kalos, "Mirror Potentials," *preprint* (1985).

[15] R. N. Grimes, R. N. Barnett, P. J. Reynolds, and W. A. Lester, Jr., "Molecular Excited States with Fixed-Node Quantum Monte Carlo" *in preparation.*

[16] M.J. Kascic Jr., *Vector Processing on the Cyber 200* (available from the Control Data Corporation)

[17] See, for example, A. Dickinson, Comp. Phys. Comm. 26, 459 (1982).

[18] W. P. Petersen, Comm. of the ACM 26, 1008 (1983)

[19] See, for example, the Cyber 205 User Guide (available from Control Data Corporation), and Cray Computer Systems Technical Note - Optimization Guide (available from Cray Research).

[20] P. J. Reynolds and W. A. Lester, Jr., Proceedings of the Cyber 200 Applications Seminar, NASA Conferance Proceedings 2295 (1983).

[21] S. Alexander, P. J. Reynolds, R. N. Barnett, and W. A. Lester, Jr., "Vectorization of Molecular Quantum Monte Carlo" *in preparation.*

[22] (a) H. Bruijnes, NMFECC Buffer 9, No. 6, 1 (1985); (b) J. Worlton, Datamation, pp 121-30, 1 September 1984.

[23] G. Rodrique, E. D. Giroux, and M. Pratt, Computer 13, 65 (1980).

[24] CDC - Cyberplus announcement, based on the AFP processor designed by Control Data Corp.

[25] J. B. Dennis, Proc. First Int. Conf. on Distributed Computing Systems, p. 430 (1979).

[26] *Interconnection Networks for Parallel and Distributed Processing,* C-L. Wu and T-Y. Feng, eds. (IEEE Comp. Society Press, Silver Spring, 1984).

[27] M. Dubois and S. Briggs, IEEE Trans Comp, C31, 1083 (1982).

[28] Physics Today, May 1984.

[29] For a detailed description of the MIDAS architecture see C. Maples and D. Logan, SIAM, *in press,* and references therein.

[30] P. J. Reynolds, D. Logan, C. Maples, and W. A. Lester, Jr., "Parallelism in Quantum Monte Carlo: Calculation of the Binding Energy of the Nitrogen Molecule" *in preparation.*

[31] D. Logan, C. Maples, D. Weaver, and W. Rathbun, Proc. 13th Int. Conf. on Parallel Processing, p. 15 (1984).

[32] A. K. Rappe, J. Comp. Chem. 5, 471 (1984).

[33] I.Y. Bucher and J.W. Moore, "Comparative Performance Evaluation of Two Supercomputers: CDC Cyber-205 and CRI Cray-1," Los Alamos Scientific Report LA-2629 (1981).

[34] V. R. Sanders and M. F. Guest, Comp. Phys. Comm. 26, 389 (1982); Martyn F. Guest and Stephen Wilson, in Supercomputers in Chemistry, Peter Lykos and I. Shavitt eds. (American Chemical Society, Washington D.C., 1981).

[35] R. A. Bair and T. H. Dunning, Jr., J. Comp. Chem. 5, 44 (1984).

[36] R. Ahlrichs, H-J. Böhm, C. Ehrhardt, P. Scharf, H. Schiffer, H. Lischka, and M. Schindler, J. Comp. Chem. 6, 200 (1985).

[37] E. Clementi, G. Corongiu, J. Detrich, S. Chin, and L. Domingo, Int. J. Quant. Chem. Symp. 18, 601 (1984).

Table 1. Comparative run times for the restructured algorithm with an ensemble size of 100. Single precision is used on all machines.

Machine	scalar seconds	vector seconds
VAX 11/780	1620.0	---
Cyber 205 (2 pipe)	82.4	16.7
Cyber 205 (4 pipe)	81.7	13.9
Cray 1S	70.5	14.5
Cray XMP[a]	49.8	9.9

[a] All calculations were done using only one processor.

Table 2. Comparative run times for the restructured algorithm with an ensemble size of 500. Single precision is used on all machines. Other than the ensemble size, all parameters are the same as in Table 1. Thus the scalar time rises by a factor of 5, but the vector times rise less rapidly.

Machine	scalar seconds	vector seconds
VAX 11/780	8100.0	---
Cyber 205 (2 pipe)	413.6	69.2
Cyber 205 (4 pipe)	415.9	43.6
Cray XMP[a]	246.7	44.4

[a] All calculations were done using only one processor.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720