Efficient Algorithm for "On-the-Fly" Error Analysis of Local or Distributed Serially Correlated Data

DAVID R. KENT IV,[1] RICHARD P. MULLER,[1]* AMOS G. ANDERSON,[1]
WILLIAM A. GODDARD III,[1] MICHAEL T. FELDMANN[2]
[1]*Materials and Process Simulation Center, Division of Chemistry and Chemical Engineering, California Institute of Technology (MC 139-74), Pasadena, California 91125*
[2]*Center for Advanced Computing Research, California Institute of Technology, Pasadena, California 91125*

Received 22 January 2004; Accepted 6 March 2007
DOI 10.1002/jcc.20746
Published online 2 May 2007 in Wiley InterScience (www.interscience.wiley.com).

Abstract: We describe the Dynamic Distributable Decorrelation Algorithm (DDDA) which efficiently calculates the true statistical error of an expectation value obtained from serially correlated data "on-the-fly," as the calculation progresses. DDDA is an improvement on the Flyvbjerg-Petersen renormalization group blocking method (Flyvberg and Peterson, J Chem Phys 1989, 91, 461). This "on-the-fly" determination of statistical quantities allows dynamic termination of Monte Carlo calculations once a specified level of convergence is attained. This is highly desirable when the required precision might take days or months to compute, but cannot be accurately estimated prior to the calculation. Furthermore, DDDA allows for a parallel implementation which requires very low communication, $O(\log_2 N)$, and can also evaluate the variance of a calculation efficiently "on-the-fly." Quantum Monte Carlo calculations are presented to illustrate "on-the-fly" variance calculations for serial and massively parallel Monte Carlo calculations.

 J Comput Chem 28: 2309–2316, 2007

Key words: Quantum Monte Carlo; serial correlation; parallel computing; variance statistic

Introduction

Monte Carlo methods are becoming increasingly important in calculating the properties of chemical, biological, materials, and financial systems. The underlying algorithms of such simulations (e.g. Metropolis algorithm[1]) often involve Markov chains. The data generated from the Markov chains are serially correlated, meaning that the covariances between data elements is non-zero. Because of this, care must be taken to obtain the correct variances for observables calculated from the data.

Data blocking algorithms to obtain the correct variance of serially correlated data have been part of the lore of the Monte Carlo community for years. Flyvbjerg and Petersen were the first to formally analyze the technique[2], but at least, partial credit should be given to Wilson[3], Whitmer[4], and Gottlieb and coworkers[5] for their earlier contributions.

We propose a new blocking algorithm, dynamic distributable decorrelation algorithm (DDDA), which gives the same results as the Flyvbjerg-Petersen algorithm but allows the underlying variance of the serially correlated data to be analyzed "on-the-fly" with negligible additional computational expense. DDDA is also ideally suited for parallel computations because only a small amount of data must be communicated between processors to obtain the global results. Furthermore, we present an efficient method for combining results from individual processors in a parallel calculation that allows fast "on-the-fly" result analysis for parallel calculations. Example calculations showing "on-the-fly" variance calculations for serial and massively parallel calculations are also presented.

All current blocking algorithms require $O(mN)$ operations to evaluate the variance m times during a calculation of N steps. DDDA only requires $O(N + m \log_2 N)$. Furthermore, current algorithms require communicating $O(N)$ data during a parallel calculation to evaluate the variance. DDDA requires only $O(\log_2 N)$. The improved efficiency permits convergence based termination in a nearly "zero cost" manner for Monte Carlo calculations.

*Present address for R.P.M.: Multiscale Computational Materials Methods, Sandia National Laboratories, Albuquerque, New Mexico 87185-1322

Correspondence to: William A. Goddard III; e-mail: wag@wag.caltech.edu

Contract/grant sponsor: Office of Scientific Computing and Office of Defense Programs; contract/grant number: DE-FGO2-97ER25308 and DOE-ASC-LLNL-B523297

Contract/grant sponsor: Fannie and John Hertz Foundation

Theory

Computer simulations of physical systems often involve the calculation of an expectation value, $\langle f \rangle$, using a complicated high-dimensional probability distribution function, $\rho(x)$.

$$\langle f \rangle \equiv \int \rho(x) f(x) dx \tag{1}$$

This expression is simple and elegant, but in many physical systems, $\rho(x)$ is too complex for eq. (1) to be useful computationally. Instead, simulations typically calculate the "time average" of f, $\bar{f}$

$$\bar{f} \equiv \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{2}$$

Here i is related to the Monte Carlo step number, and x_i is sampled from the distribution $\rho(x)$. Then, assuming ergodicity, $\langle f \rangle$ and $\bar{f}$ can be related through

$$\langle f \rangle = \lim_{n \to \infty} \bar{f} = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{3}$$

On modern computers, very large samplings are used to approach this limit. However, since such sampling is necessarily always finite, $\bar{f}$ will fluctuate as the calculation progresses because it has a non-zero variance, $\sigma^2(\bar{f})$. This variance can be expressed as

$$\sigma^2(\bar{f}) = \frac{1}{n^2} \sum_{i,j=1}^{n} [\langle f(x_i) f(x_j) \rangle - \langle f(x_i) \rangle \langle f(x_j) \rangle] \tag{4}$$

$$= \frac{\sigma^2(f)}{n} + \frac{2}{n^2} \sum_{i=1}^{n} \sum_{j>i}^{n} \mathrm{cov}(f(x_i), f(x_j)) \tag{5}$$

When the $\{f(x_i)\}$ are uncorrelated, the covariance terms are zero, and eq. (5) reduces to the typical variance relation.

$$\sigma^2(\bar{f}) = \frac{\langle f^2 \rangle - \langle f \rangle^2}{n} = \frac{\sigma^2(f)}{n} \tag{6}$$

Calculations which use Markov chains to generate $\{x_i\}$, such as Metropolis algorithm[1] based calculations, produce $\{f(x_i)\}$ with non-zero covariances. This results because the probability of picking x_i is dependant on the value of x_{i-1}. If eq. (6) is used to calculate the variance of such systems, the result will be incorrect because the covariances between samples are not included.

Without loss of generality, eqs. (2) and (5) can be expressed in terms of the random variate x_i instead of the random variate $f(x_i)$. This gives

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{7}$$

$$\sigma^2(\bar{x}) = \frac{1}{n^2} \sum_{i,j=1}^{n} \gamma_{i,j} \tag{8}$$

where $\gamma_{i,j} = \mathrm{cov}(x_i, x_j)$.

Then, if it is assumed that a Markov chain method with stationary transition probabilities, such as Monte Carlo or molecular dynamics at equilibrium, was used to generate $\{x_i\}$.

$$\sigma^2(\bar{x}) = \frac{1}{n} \xi_0 + \frac{2}{n} \sum_{t=1}^{n-1} (n-t) \xi_t \tag{9}$$

where ξ_t is the covariance between data points t steps apart.

$$\xi_t \equiv \gamma_{i,j} \quad t = |i-j| \tag{10}$$

In this representation, it is possible to define a blocking transform that takes $\{x_i\} \rightarrow \{x'_i\}$.

$$x'_i = \frac{1}{2} \{x_{2i-1} + x_{2i}\} \tag{11}$$

$$n' = \frac{1}{2} n \tag{12}$$

In performing this transform, it can be shown[2] that

$$\bar{x}' = \bar{x} \tag{13}$$

$$\sigma^2(\bar{x}') = \sigma^2(\bar{x}) \tag{14}$$

$$\xi'_t = \begin{cases} \frac{1}{2}\xi_0 + \frac{1}{2}\xi_1 & \text{for } t = 0 \\ \frac{1}{4}\xi_{2t-1} + \frac{1}{2}\xi_{2t} + \frac{1}{4}\xi_{2t+1} & \text{for } t > 0 \end{cases} \tag{15}$$

Furthermore, from eq. (9), we see

$$\sigma^2(\bar{x}) \geq \frac{\xi_0}{n} \tag{16}$$

and from Eqs. (12) and (15), it can be shown that ξ_0/n increases as blocking transforms are applied, unless $\xi_1 = 0$, in which case ξ_0/n is invariant. Further analysis shows that with repeated application of the blocking transforms in Eqs. (11) and (12) a fixed point is reached where $\sigma^2(\bar{x}) = \xi_0/n$. Therefore, the variance of a data set can be evaluated by performing blocking operations until ξ_0/n remains constant with further blocking operations.

During a calculation, χ can be used to estimate ξ_0/n.

$$\chi = \frac{\frac{1}{n}(\sum_{i=1}^{n} x_i^2) - \frac{1}{n^2}(\sum_{i=1}^{n} x_i)^2}{n-1} \tag{17}$$

When enough blocking transforms have been applied to reach the fixed point, the blocked variables are independent Gaussian random variables making χ also a Gaussian random variable with standard deviation $\chi\sqrt{2/(n-1)}$.

The above analysis deals with serially correlated data from Markov processes. Branching processes, such as diffusion or Green's function QMC, also generate data that have parallel correlation. This can be removed by averaging the data from each iteration to make new data elements[6]. These new data elements are still serially correlated and must then be analyzed appropriately to obtain the true variance of the calculation.

Algorithms

Flyvbjerg-Petersen Algorithm

The Flyvbjerg-Petersen algorithm[2] is conceptually very simple. The average, $\bar{x}$, and χ, for the data, $\{x_i\}$, are calculated using Eqs. (7) and (17). A new blocked data set is generated from this

Table 1. Comparison of Computational Costs.

	Flyvbjerg-Petersen algorithm	Dynamic distributable decorrelation algorithm (DDDA)
Operations	$O(mN)$	$O(N + m \log_2 N)$
State size	$O(N)$	$O(\log_2 N)$
Parallel communications	$O(N)$	$O(\log_2 N)$
Parallel variance evaluation	$O(N)$	$O(\log_2 N \log_2 P)$

N is the number of data points analyzed, m is the number of times the variance is evaluated during a calculation, and P is the number of processors.

data using the block transforms described in Eqs. (11) and (12). The average and χ of these data are then evaluated. This process is repeated until no more blocking operations can be performed. The true variance is the value of χ obtained when further blocking operations do not change the value.

Overall, this algorithm requires $O(N)$ operations, where N is the number of unblocked data points, to evaluate the true variance of the calculation. The state of the algorithm is given by an array of all unblocked data elements and is therefore of size $O(N)$. For many calculations, N is very large ($>10^9$) forcing the state to be saved to disk because it does not fit in the machine's RAM. Because of this, an additional slow $O(N)$ cost is often incurred from reading the data in from disk in order to analyze it.

The Flyvbjerg-Petersen algorithm is an inherently serial algorithm. To use it for a parallel calculation, all of the unblocked data must be sent to one processor where it is concatenated and analyzed as above. Such an operation requires an $O(N)$ communication, where N is the number of unblocked data elements. Furthermore, the entire burden of error analysis is placed on one processor, making the variance calculation expensive for very large samplings. Also, the large amount of data communicated to one processor can potentially saturate the bandwidth available to this processor.

During a stochastic simulation, it is desirable to evaluate the variance of calculated quantities periodically to determine when the calculation is converged and can be terminated. If the variance is to be evaluated m times during the calculation, the Flyvbjerg-Petersen algorithm requires $O(mN)$ operations, to accomplish this. This can be prohibitively expensive for large N or m.

A summary of the computational costs is listed in Table 1.

Dynamic Distributable Decorrelation Algorithm (DDDA)

The equations implemented by DDDA are *exactly* the same as those presented by Flyvbjerg and Petersen. DDDA is a new algorithm to evaluate these equations. The new algorithm involves two classes:

1. Statistic Class (*Pseudocode is listed in Appendix A*)
   The Statistic class stores the number of samples, n, running sum of x_i, and running sum of x_i^2 for *the data that is entered into it,* $\{x_i\}$. This allows straightforward calculation of the average, $\bar{x}$, [eq. (7)] and χ [eq. (17)].
2. Decorrelation Class (*Pseudocode is listed in Appendix B*)
   The Decorrelation class stores a vector of Statistic objects, where the ith element of the vector corresponds to data that has been partitioned into blocks 2^i long, and a collection of data samples waiting to be added to the vector. The variance and average for the ith element of the vector can be evaluated by calling the corresponding functions in the appropriate Statistic object.

As data is generated during the calculation, it is added to a Decorrelation object. It is first added to the 0th element of the vector of Statistic objects (vectors numbered from 0) If there is no sample waiting on the 0th level, this sample is stored as the waiting sample for the 0th level; otherwise, this sample and the waiting sample for the 0th level are averaged to create a new sample, and the waiting sample is removed. This new sample is then added to the 1st level in the same fashion as above. This process repeats until a level is reached with no waiting samples. By adding data this way, new data blocks are generated as soon as there is enough data to create them. Furthermore, because the newly generated data blocks are added to Statistic objects as they are generated, the variance for a particular block size can be immediately evaluated with *very* few operations. Using these data, it is straightforward to evaluate the true variance as is done with standard blocking methods.

During a parallel calculation, each processor will have a Decorrelation object to which it adds data. The global results are then obtained by combining the Decorrelation objects from each processor into a global Decorrelation object. This can be accomplished using a binary operator to add two Decorrelation objects together. The first step in this process adds the Statistic vectors, from the two Decorrelation objects, element by element to form a new Statistic vector. Then, beginning with the 0th level, the waiting samples are combined to create either new waiting samples or new averaged samples to be added to the new Statistic vector and combined with waiting samples from the next higher level. Evaluating this binary addition requires only $O(\log_2 N)$ operations, where N is the number of samples.

Analysis of DDDA

The equations implemented by DDDA are *exactly* the same as those presented by Flyvbjerg and Petersen; both require $O(N)$ operations to evaluate the variance of N data samples. In contrast to Flyvbjerg and Petersen, the state (minimal set of data an algorithm must store) of DDDA is only of size $O(\log_2 N)$. The small size of this state ($\log_2 10^9 \approx 30$) means that all necessary data can be stored in RAM, avoiding the read-in expense often encountered with the Flyvbjerg-Petersen algorithm. Also, the small state yields a very small checkpoint from which calculations can be restarted. If an upper bound is known on the block size, then the algorithm can be modified slightly to give a state size of only $O(1)$.

One major advantage of DDDA over the Flyvbjerg-Petersen algorithm, is its ability to efficiently evaluate the true variance of a calculation "on-the-fly." If the variance is to be evaluated m times during the calculation, the Flyvbjerg-Petersen algorithm requires

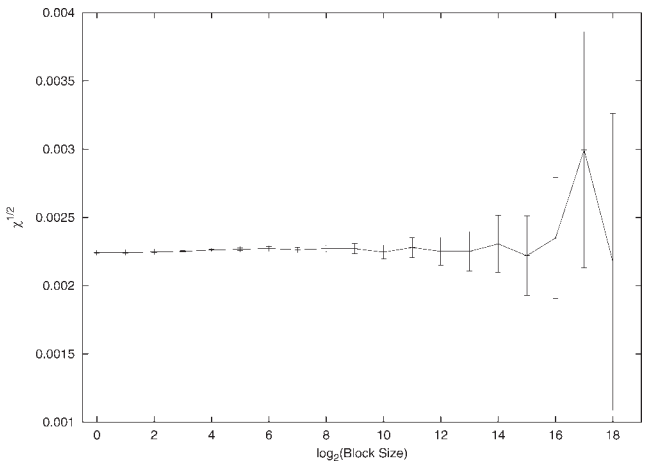

Figure 1. $\sqrt{\chi}$ [eq. (17)] as a function of block size for a variational QMC "particle-in-a-box" calculation using uncorrelated data points (Method 1). The Flyvbjerg-Petersen algorithm and DDDA yield exactly the same results.

$O(mN)$ operations to accomplish this while DDDA requires only $O(N + m \log_2 N)$. The improved computational complexity makes convergence based termination practical to implement.

The other major advantage of DDDA over the Flyvbjerg-Petersen algorithm is its performance on parallel calculations. Because the state of DDDA is so compact, only $O(\log_2 N)$ data elements must be communicated between processors. Furthermore, because two Decorrelation objects can be added with $O(\log_2 N)$ operations, a binary tree can be used to evaluate the global variance of the parallel calculation in only $O(\log_2 N \log_2 P)$ operations, where P is the number of processors. The expense of the additions is distributed over a large number of processors. This low complexity evaluation makes possible "on-the-fly" variance determination for massively parallel calculations.

A summary of the computational costs is listed in Table 1.

Computational Experiments

"On-the-Fly" Variance Determination for a Single Processor Variational QMC Particle-in-a-Box Calculation

To illustrate DDDA, we use variational quantum Monte Carlo (VMC)[7] to calculate the energy for a one-dimensional particle-in-a-box of length one. For this illustration, we use a normalized trial wavefunction

$$\Psi_T = \sqrt{30}(x - x^2) \tag{18}$$

to approximate the exact ground state wavefunction, $\Psi_{\text{Exact}} = \sqrt{2}\sin(\pi x)$. The expected energy of the system is given by

$$\langle E \rangle = \int_0^1 \Psi_T \hat{H} \Psi_T dx = \int_0^1 \Psi_T^2 \left(\frac{\hat{H}\Psi_T}{\Psi_T} \right) dx = \int_0^1 \rho_T(x) E_L(x) dx, \tag{19}$$

where $\hat{H}$ is the Hamiltonian for the system, $E_L(x)$ is the local energy, and $\rho T(x)$ is the probability distribution of the particle. Since the Ψ_T is not an eigenfunction for this system, the local energy will not be constant and the calculated energy expectation value will fluctuate as the calculation progresses.

Equation (19) can be evaluated in two ways:

- One option (Method 1) is to generate points distributed with respect to $\rho T(x)$ by directly inverting[8] $\rho T(x)$ and use these points to sample $E_L(x)$. Because $\rho T(x)$ is directly inverted, this method will produce uncorrelated data.
- A second option (Method 2) is to generate points distributed with respect to $\rho T(x)$ using the Metropolis algorithm[1] and use these points to sample $E_L(x)$. Because the Metropolis algorithm employs a Markov chain, this method will produce serially correlated data.

Performing 10^6 Monte Carlo steps gives expected energy values of 4.9979(23) for Method 1 and 4.9991(59) for Method 2. Both values agree with the analytic value of 5. We should also note that the error estimates of the correlated and uncorrelated calculations are different. These error estimates illustrate that serially correlated data does not provide as much information as uncorrelated data, resulting in a larger standard deviation for the correlated case (Method 2) than the uncorrelated case (Method 1) when using the same number of samples.

Figures 1 and 2 show the calculated standard deviation vs. block size for uncorrelated (Method 1) and correlated (Method 2) VMC calculations.

In both cases, the plateau in the plot corresponds to the true standard deviation value. Fluctuations associated with large block sizes result from dividing the data into a small number of blocks making the data very noisy.

Evaluating the standard deviation in the correlated VMC calculation without data blocking yields an estimate of the standard deviation that is much too small. This corresponds to $log_2(BlockSize) = 0$ in Figure 2 and illustrates the potential dangers in

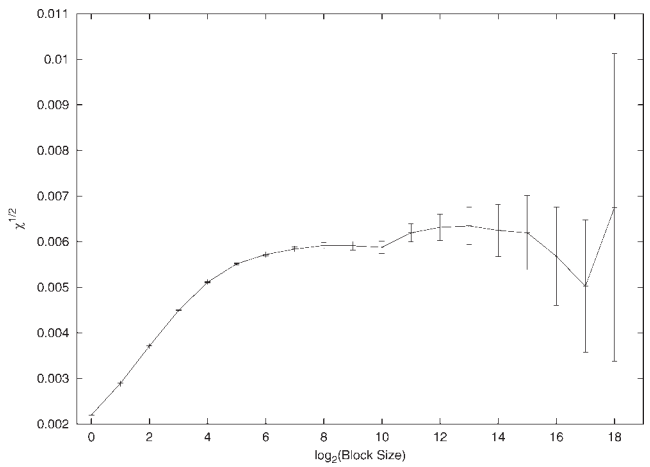

Figure 2. $\sqrt{\chi}$ [eq. (17)] as a function of block size for a variational QMC "particle-in-a-box" calculation using serially correlated data points (Method 2). The Flyvbjerg-Petersen algorithm and DDDA yield exactly the same results.

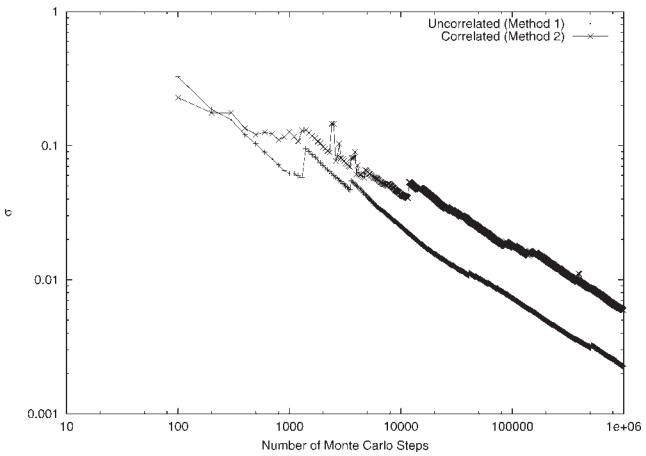

Figure 3. Standard deviation as a function of number of Monte Carlo steps for a variational QMC "particle-in-a-box" calculation. The standard deviation was evaluated "on-the-fly" using DDDA.

reporting error estimates without accounting for the serial correlation that may exist in the data.

The ability of DDDA to evaluate the standard deviation "on-the-fly" for a single processor calculation is demonstrated in Figure 3. During the VMC particle in a box calculations, the standard deviation was evaluated every 100 Monte Carlo steps.

"On-the-Fly" Variance Determination for a Massively Parallel Variational QMC Calculation of RDX

To illustrate the ability of DDDA to evaluate the variance from a large parallel Monte Carlo calculation "on-the-fly," we performed a series of 1024 processor massively parallel variational quantum Monte Carlo (VMC) calculations on the high explosive material RDX (Fig. 4), cyclic $[CH_2—N(NO_2)]_3$.

Of the three calculations performed, one was the ground state structure, and the other two were unimolecular decomposition transition states for the concerted dissociation and $N—NO_2$ bond fission reactions. Geometries of the species were obtain from previous DFT calculations on the system[9].

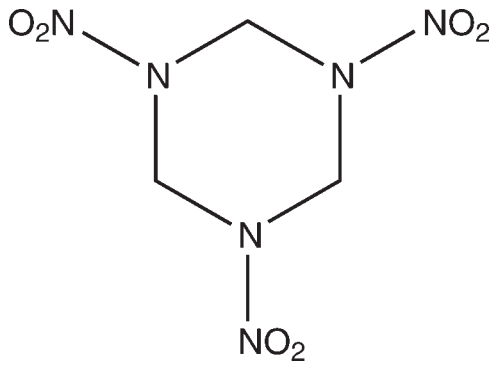

Figure 4. The RDX molecule, cyclic $[CH_2—N\ (NO_2)]_3$.

Table 2. Pade-Jastrow Correlation Function Parameters for RDX.

	a	*b*
$u\uparrow\downarrow$	0.5	3.5
$u\uparrow\uparrow$, $u\downarrow\downarrow$	0.25	100
$u\uparrow H$, $u\downarrow H$	−1	100
$u\uparrow C$, $u\downarrow C$	−6	100
$u\uparrow N$, $u\downarrow N$	−7	100
$u\uparrow O$, $u\downarrow O$	−8	100

The VMC wavefunction, Ψ_{VMC}, used for the calculation is the product of a Hartree-Fock wavefunction, Ψ_{HF}, and a Pade-Jastrow correlation function, $J_{Corr.}$

$$\Psi_{\mathrm{VMC}} = \Psi_{\mathrm{HF}} J_{\mathrm{Corr}} \tag{20}$$

$$J_{\mathrm{Corr}} = \exp\left(\sum_i \sum_{j<i} u_{i,j}\right) \tag{21}$$

$$u_{i,j} = \frac{a_{i,j} r_{i,j}}{1 + b_{i,j} r_{i,j}} \tag{22}$$

Ψ_{HF} was calculated using Jaguar[10,11] with a 6–31G** basis set[12]. The Pade-Jastrow parameters (Table 2) where chosen to remove singularities in the local energy. Furthermore, they maintain the structure of the Hartree-Fock wavefunction everywhere except where two particles closely approach one another. Though much work has been done on wavefunction optimization techniques[7,13–21], we do not optimize the Pade-Jastrow parameters because this calculation is to demonstrate DDDA and not to obtain a high-accuracy VMC energy, which would require parameter optimization.

Calculations were performed on the ASCI Nirvana supercomputer at the Los Alamos National Laboratory using 1024 MIPS 10,000 processors running at 250 MHz. Each calculation took approximately 8 hours to complete and was composed of roughly 3×10^7 Monte Carlo steps. Of the three calculations, two were run to completion while the third calculation was stopped a fraction of the way through the run and restarted from checkpoints to verify the ease and efficiency with which these new data structures allow for checkpointing of the program state variables. The RDX calculations successfully completed inde-

Table 3. Total Energies (Hartree) for the Various Calculations on RDX.

RDX species	Hartree fock	Variational quantum Monte Carlo
Ground state	−892.491	−893.35 (4)
Concerted dissociation	−892.369	−893.29 (5)
$N - NO_2$ bond fission	−892.259	−893.20 (4)

The HF results were obtained from Jaguar 4.1 with the 6-31G** basis set Variational Quantum Monte Carlo based on 3×10^7 points.

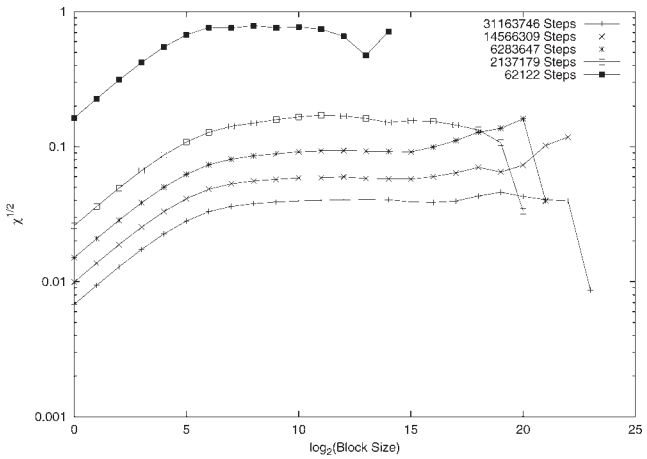

Figure 5. Evolution of $\sqrt{\chi}$ as a function of block size as a variational QMC calculation of the RDX groundstate progresses. Shown here are the results for five cases with 6.2×10^4, 2.1×10^6, 6.2×10^6, 1.5×10^7, and 3.1×10^7 total Monte Carlo steps.

pendent of whether they were run to completion or checkpointed and restarted.

Energies for the Hartree Fock, and variational quantum Monte Carlo[22] calculations are presented in Table 3. The VMC energies are presented for completeness and should not be taken to be highly accurate energies because the variational parameters have not been optimized.

Figures 5 and 6 show the evolution of the standard deviation of the total energy for three different RDX species *as the Monte Carlo calculations progress*. In Figure 5, we see that the plateau in the plot of standard deviation vs. block size, indicating the true variance, is reached for a block size of roughly 2^8. Results from the RDX transition state structures are similar and require a block size of 2^8 to 2^{13}, depending on the system. Figure 6 shows the standard deviations evaluated "on-the-fly" for the massively parallel calculations. These values are found to decrease roughly with the square root of the number of samples, as is expected.

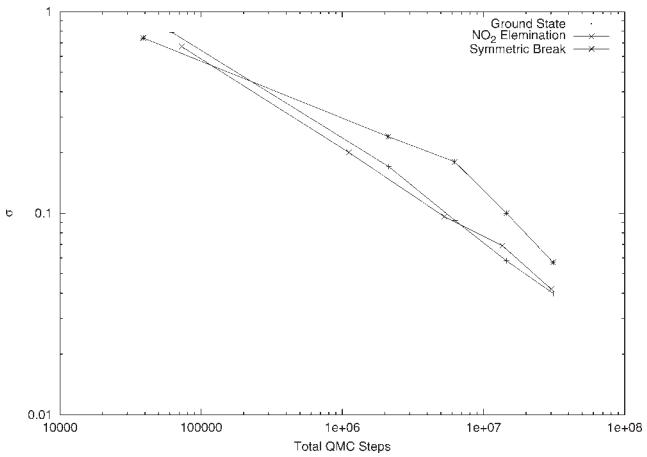

Figure 6. Standard deviation as a function of number of Monte Carlo steps for a 1024 processor variational QMC calculations of RDX. The standard deviation was evaluated "on-the-fly" using DDDA.

Conclusions

The above analysis has shown that DDDA is significantly more efficient than standard blocking algorithms at evaluating the variance of a quantity multiple times, "on-the-fly", during a calculation ($O(N + m)$ vs. $O(mN)$). Additionally, the state needed to checkpoint the calculation or evaluate the variance in a parallel calculation is only $O(\log_2 N)$ for DDDA and $O(N)$ for current algorithms. This leads to smaller checkpoints and significantly less communications for parallel calculations. The small state size will facilitate calculations on computational grids where many processors are used but bandwidth is limited.

Because DDDA efficiently evaluates the variance "on-the-fly" for both serial and parallel calculations, it is now possible to use an efficient convergence based termination scheme. Instead of prespecifying the number of data points a calculation will use, points are gener- ated until the observed quantities are converged to the specified tolerance. This eleminates calculations terminating before they are completed or running too long and wasting computational resources. Additionally, specifying a desired level of convergence is much more natural than specifying the number of Monte Carlo steps for a non-expert user or an expert user analyzing a new system.

Often when blocking is used for error analysis, the data is *preblocked* before it is analyzed. This consists of blocking the data before any data analysis takes place. Because the correct block size is not known *a priori*, the Flyvbjerg-Petersen algorithm must, then be used to analyze the preblocked data. Preblocking does reduce the amount of data that must be stored, analyzed, and communicated, but it does not change the complexity of the computational costs of the Flyvbjerg-Petersen algorithm (Table 1) making it inferior to DDDA. It is possible to preblock and then use DDDA, but this is not necessary. Because the storage and communication costs of DDDA are $O(\log_2 N)$, reducing N by a constant factor makes only a small change in the state size negating the benefits of preblocking.

Acknowledgments

This research was supported by the DOE-ASCI and DOE-ASC grants at Caltech.

We thank Sharon Brunett of the California Institute of Technology's Center for Advanced Computational Research for assistance with the computations on the ASCI-BLUE Mountain supercomputer at Los Alamos National Laboratory.

The computational resources at the MSC were provided by the NSF (CHE-99MRI), IBM (SUR Grant), and ARO-DURIP (1997). The computational resources at LANL were under the DOE ASCI ASAP project at Caltech. Other support for the MSC came from NIH, Chevron-Texaco, 3M, Avery-Dennison, Dow, GM, Seiko-Epson, Beckman Institute, Asahi Chemical, and Nippon Steel.

Appendix A: Statistic Class Pseudocode

```
1. Pseudocode for Statistic.initialize()
    # Initialize a new instance of the Statistic class
    Statistic.initialize()
      NSamples = 0.0
      Sum = 0.0
      SumSq = 0.0
2. Pseudocode for Statistic.addData(new_sample)
    # Add a new data element to this Statistic object
    Statistic.addData(new_sample)
      NSamples = NSamples + 1
      Sum = Sum + new_sample
      SumSq = SumSq + new_sample*new_sample
3. Pseudocode for Statistic.addition(A, B)
    # Add two Statistic objects and return the result
    Statistic.addition(A,B)
      C=new Statistic()
      C.NSamples = A.NSamples + B. NSamples
      C.Sum = A.Sum + B.Sum
      C.SumSq = A. SumSq + B.SumSq
      return C
```

Appendix B: Decorrelation Class Pseudocode

```
1. Pseudocode for Decorrelation.initialize()
    # Initialize a new instance of the Decorrelation class
    Decorrelation.initialize():
      Size = 0
      NSamples = 0
      BlockedDataStatistics = [new Statistic()]
      waiting_sample = [0]
      waiting_sample_exists = [false]
2. Pseudocode for Decorrelation.addData(new_sample)
    # Add a new data element to this Decorrelation object
    Decorrelation.addData(new_sample):
      NSamples = NSamples + 1
    # Lengthen the vectors, when necessary, to accommodate
    all enterec data
    if NSamples > = 2^Size:
      Size = Size + 1
      BlockedDataStatistics =
        BlockedDataStatistics.append(new Statistic ())
      waiting_sample = waiting_sample.append(0)
      waiting_sample_exists = waiting_sample_exists.
        append(false)
      BlockedDataStatistics[0].add_Data(new_sample)
      carry = new_sample
      i = 1
      done = false
    # Propagate the new sample up through the data structure
    while (not done):
      if waiting_sample_exists[i]:
        new_sample = (waiting_sample[i] + carry)/2
        carry = new_sample
        BlockedDataStatistics[i].addData(new_sample)
        waiting_sample_exists[i] = false
      else:
        waiting_sample_exists[i] = true
        waiting_sample[i] = carry
        done = true
      i = i+1
      if i > Size:
        done = true
3. Pseudocode for Decorrelation.addition(A,B)
    # Add two Decorrelation objects and return the result
    Decorrelation.addition(A,B):
      C = new Decorrelation()
      C.NSamples = A.NSamples + B.NSamples
    # Make C big enough to hold all the data from A and B
    while C. NSamples > = 2C.Size:
      C.Size = C.Size + 1
      C.BlockedDataStatistics =
        C. BlockedDataStatistics.append(new Statistic())
      C.waiting.sample = C. waiting_sample.append(0)
      C.waiting.sample_exists =
        C. waiting_sample_exists.append(false)
      carry_exists = false
      carry = 0
      for i from 0 to C.Size-1:
        if i < = A.Size:
          StatA = A.BlockedDataStatistics[i]
          waiting_sampleA = A.waiting_sample[i]
          waiting_sample_existsA = A.waiting_sample_exists[i]
        else:
          StatA = new Statistic()
          waiting_sampleA = 0
          waiting_sample_existsA = false
        if i < = B.Size:
          StatB = B.BlockedDataStatistics[i]
          waiting_sampleB = B.waiting_sample[i]
          waiting_sample_existsB = B.waiting_sample_exists[i]
        else:
          StatB = new Statistic()
          waiting_sampleA = 0
          waiting_sample_existsA = false
        C.BlockedDataStatistics[i] =
          C.BlockedDataStatistics[i].addition(StatA, StatB)
        if (carry_exists & waiting_sample_existsA &
        waiting_sample_existsB):
          # Three samples to handle
          C.BlockedDataStatistics[i].addData(
            (waiting_sampleA+waiting_sampleB)/2)
          C.waiting_sample[i] = carry
          C waiting_sample_exists[i] = true
          carry_exists = true
          carry = (waiting_sampleA+waiting_sampleB)/2
        else if (not carry_exists & waiting_sample_existsA &
          waiting_sample_existsB):
          # Two samples to handle
          C. BlockedDataStatistics[i].addData(
            (waiting_sampleA+waiting_sampleB)/2)
          C.waiting_sample[i] = 0
          C.waiting_sample_exists[i] = false
          carry_exists = true
```

```
        carry = (waiting_sampleA+waiting_sampleB)/2
      else if (carry_exists & not waiting_sample_existsA &
        waiting_sample_existsB):
        # Two samples to handle
        C.BlockedDataStatistics[i].addData(
          (carry+waiting_sampleB)/2)
        C.waiting_sample[i] = 0
        C.waiting_sample_exists[i] = false
        carry_exists = true
        carry = (carry+waiting_sampleB)/2
      else if (carry_exists & waiting_sample_existsA &
        not waiting_sample_existsB):
        # Two samples to handle
        C.BlockedDataStatistics[i].addData(
          (carry+waiting_sampleA)/2)
        C.waiting_sample[i] = 0
        C.waiting_sample_exists[i] = false
        carry_exists = true
        carry = (carry+waiting_sampleA)/2
      else if (carry_exists or waiting_sample_existsA or
        waiting_sample_existsB):
        # One sample to handle
        C.waiting_sample[i] = carry +
          waiting_sampleA + waiting_sampleB
        C.waiting_sample_exists[i] = true
        carry_exists = false
        carry = 0
      else:
        # No samples to handle
        C.waiting_sample[i] = 0
        C.waiting_sample_exists[i] = false
        carry_exists = false
        carry = 0
    return C
```

Appendix C: Simple Example Calculation Pseudocode

```
for all processors
  # Initialize error analysis data structure for each processor
  LocalErrorAnalysisDataStructure = new Decorrelation()
  while generating new data points:
    # Generate new data and add it to the local error
    # analysis data structure
    new_data = generateNewDataPoint()
    LocalErrorAnalysisDataStructure.addData(new_data)
  if want global results:
    Obtain the global results for the calculation with a binary tree
      parallel reduction operation using Decorrelation.add(...)
      to add LocalErrorAnalysisDataStructure from each processor
```

References

1. Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E. J Chem Phys 1953, 21, 1087.
2. Flyvberg, H.; Peterson, H. J Chem Phys 1989, 91, 461.
3. Wilson, K. In Recent Developments in Gauge Theories; Hoof, e. a. G. 't Ed.; Plenum: New York, 1979.
4. Whitmer, C. Phys Rev D 1984, 29, 306.
5. Gottlieb, S.; Mackenzie, P.; Thacker, H.; Weingarten, D. Nuclear Phys B 1986, 263, 704.
6. Rothstein S. M.; Vrbik J. J Comput Chem 1988, 74, 127.
7. Umrigar, C. J. In Quantum Monte Carlo Methods in Physics and Chemistry; Nightingale, M. P.; Umrigar, C. J.; Eds.; Kluwer Academic: Dordrecht, The Netherlands. 1999; Vol. 525 of Nato Science Series C: Mathematical and Physical Sciences, pp. 129–160.
8. Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; Vetterling, W. T. Numerical Recipes in C: The Art of Scientific Computing; Cambridge University Press: New York, 1993.
9. Chakraborty, D.; Muller, R. P.; Dasgupta, S.; III, W. G. J Phys Chem 2000, 104, 2261.
10. Ringnalda, M. N.; Langlois, J.-M.; Murphy, R. B.; Greeley, B. H.; Cortis, C.; Russo, T. V.; Marten, B.; Donnelly, R. E. Jr.; Pollard, W. T.; Cao, Y.; Muller R. P.; Mainz, D. T.; Wright, J. R.; Miller, G. H.; Goddard, W. A. III.; Friesner, R. A. Jaguar v4.1; Schrödinger, Inc.: Portland, Oregon, 2001.
11. Greeley, B. H.; Russo, T. V.; Mainz, D. T.; Friesner, R. A.; Goddard, W. A. III.; Donnelly, R. E. Jr.; Ringnalda, M. N. J Chem Phys 1994, 101, 4028.
12. Hehre, W. J.; Ditchfield, R.; Pople, J. A. J Chem Phys 1972, 56, 2257.
13. Umrigar, C.; Wilson, K.; Wilkins, J. Phys Rev Lett 1988, 60, 1719.
14. Mitas, L.; Grossman, J. Abstr Pap Am Chem Soc 1996, 211, 21.
15. Mitas, L. Phys B 1997, 237, 318.
16. Grossman, J.; Mitas, L. Phys Rev Lett 1997, 79, 4353.
17. Foulkes, W.; Mitas, L.; Needs, R.; Rajagopal, G. Rev Mod Phys 2001, 73, 33.
18. Kwon, Y.; Ceperley, D.; Martin, R. Phys Rev B 1996, 53, 7376.
19. Ceperley, D. M.; Mitas, L. In Advances in Chemical Physics; Prigogine, I.; Rice, S. A.; John Wiley & Sons, Inc.: New York, 1996; Vol. XCIII.
20. Fahy, S.; Wang, X.; Louie, S. Phys Revi B 1990, 42, 3503.
21. Fahy, S.; Wang, X.; Louie, S. Physi Rev Lett 1988, 61, 1631.
22. Feldmann, M. T.; IV. D. R. K. QM^{c}Beaver v20020109 ©. SourceForge: GNU Public License (2001–2002), quantum Monte Carlo open-source software (http://sourceforge.net/projects/qmcbeaver). Available at http://sourceforge.net/projects/qmcbeaver.