Load Balancing Performance in a Quantum Monte Carlo Method *

Paul A. Farrell, Xiaodong Hu, Michael A. Lee, Irina V. Lomonosov
Mathematics and Computer Science Department of Physics
Kent State University, Kent, Ohio 44242.

1 Abstract

We have developed and tested algorithms for use in Quantum Monte Carlo simulations in a parallel environment using MPI. The difficulty in developing an efficient parallel algorithm lies in the randomness of the size of the computational tasks distributed to the processors. We report here on characterization of the random load and its impact on the performance of an elementary load balancing strategy.

2 Introduction

There are several related algorithms which are collectively referred to as Quantum Monte Carlo methods. Generally they share the characteristic of being nondeterministic and having a degree of unpredictability in the computational work needed for a given simulation run. The particular method with which we have experience is the Greens Function Monte Carlo method. It is used to solve problems in quantum many-body physics. The technique is quite mature[3] and has evolved so that it is able to address problems in many areas. Here, we limit our attention to systems of interest to quantum chemistry[1, 4].

3 The Computational Task and the Origin of Random Load

The quantum many-body problem for N particles in three dimension is a partial differential equation in 3N dimensions. The problem of solving the Schrödinger equation may be reformulated as a 3N dimensional Fredholm integral,

$$f^{(n+1)}(R) = E \int K(R, R') f^{(n)}(R') dR'. \quad (1)$$

which the the Greens Function Monte Carlo method solves by iteration for the smallest positive eigenvalue. The unknown function in our problem is f(R), defined on a real 3N dimensional space. The values of the variable R actually correspond to the positions of the N quantum particles. Unfortunately the kernel, $K(R, R')$, lacks the important property of being normalized to unity. It and the unknown functions do have the essential property that they are positive definite, which permits their use as probability distribution functions in the Monte Carlo procedures.

Our computational task can be stated simply. Beginning from reasonable guesses for the energy, E, and the unknown function, $f^{(0)}(R)$, carry out the iteration of Eq. 1 to convergence. Determine the eigenvalue by the condition that it is the smallest proportionality constant that makes the output equal to the input after convergence. In the next section we describe the use of the Quantum Monte Carlo method to perform these integrals.

3.1 Monte Carlo Integration

Because the desired solution, $f(R)$, can be shown to be positive definite, we may choose to normalize it to have unit integral and then treat is as a probability distribution function. The Monte Carlo integration is performed by generating a set of real-space positions, $\{R_i\}$ which are sampled from the function $f^{(n)}(R)$. Initially, this may be a known analytic function for $f^{(0)}(R)$, but subsequent iteration only give the values of the coordinate points themselves. In the simulation, this may be thousands of points. In the language of this method, a single R_i is a **configuration** and the set $\{R_i\}$ is an **ensemble** of configurations. The n-th iteration produces the n-th **generation** of the ensemble. The converged ensemble is the culmination of the calculation. The process of sampling the kernel $EK(R, R')$ may yield one, zero or several values of R_i, all of which become members of the next generation ensemble. This is the essential randomness

*This work supported in part through the Ohio Supercomputer Center under grants PGS030 and PGS191, by the PhAROh MRA alliance, and by NSF CDA 9617541.

To appear in the Proceedings of the Midwest Workshop on Parallel Processing (MWPP), August 11–13, 1999, Kent State University, published by the Association of Computing Machinery (ACM)

of the method and the origin of the load balancing issue in the parallel implementation of the calculation.

In any given iteration of Eq. 1, there is a set of configurations corresponding to the ensemble for the n-th generation which we will denote by $\{R'_i\}$. From these, the ensemble for the n+1 generation is to be created. Call them $\{R_i\}$. These two sets of configurations are stored on stacks.

We may describe the algorithm in the following manner:

1. Initialize a large number of configurations $\{R'_i\}$ on a stack.
2. For each R'_i, call the procedure to sample the kernel, $E\ K(R, R'_i)$
3. When the old stack is empty, the iteration is complete. Interchange the old and new stacks and restart step 2.

The simulation ends when a predetermined number of iterations have been completed. The number of iterations is determined by the accuracy desired in the physical quantities being simulated, since the statistical errors decrease with the square root of the number of iterations it is desireable to do many iterations.

4 Development of the Parallel Algorithm

The timing of the operations performed in the Greens Function Monte Carlo depend greatly on the particular problem being performed. Our example is the electronic structure of the lithium hydride molecule. This system has only four electrons. While LiH is a small molecule, its electronic structure has been the subject of investigation more than any other molecular system. It is thus an excellent benchmark.

The obvious granular division is to take the entire ensemble and distribute it evenly over all available processors. Clearly, there is no reason to redistribute all configurations at every iteration. There is some small overhead in the calculation and monitoring of physical quantities of interest during the course of the calculation, but even this monitoring and communication is needed only every hundred iterations or so.

The above considerations lead to a parallel strategy. Take a few thousand configurations, distribute them among a few hundred processors. Let each processor operate independently for a few tens of iterations and then collect results. Sec. 5 describes the quantitative characterization of the number of ensembles on each processor as the calculation proceeds and the resulting load balancing problem.

5 Load Balancing Investigations

The initial parallel implementation is easily implemented. A modest size ensemble was distributed evenly over all processors and propagated for several generations. We first used this simple implementation to track the changes in the number of configurations on each processor.

5.1 Characterizing the Random Load

The basic problem caused by the fluctuation of the number of configurations on the processors can be seen after only a single iteration of our equation. In Fig. 1, we show the results of distributing 4096 configurations over 64 processors, each processor initially getting 64 configurations, and running the parallel algorithm for a single generation. After just one generation, some processors have nearly twice as many configurations as other processors. In the second iteration then, some processors are provided with twice as much work to perform as the others. The fluctuations of the number of configurations in an ensemble are the same size in the second step as the first and the histogram broadens rapidly as the number of iterations increases. It is clear from this plot that the disparity in the number of configurations due to this effect will have a major effect on the relative load on processors, and needs to be addressed in any load balancing strategy.

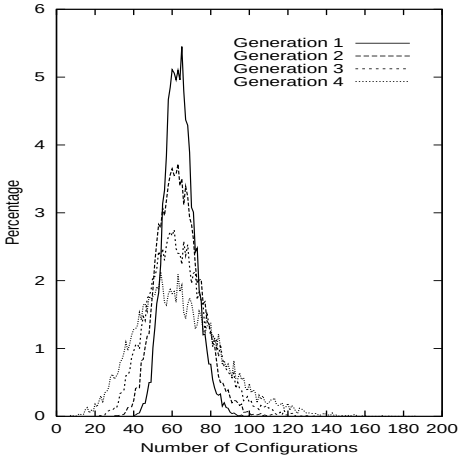

Figure 1: Number of configurations on 64 parallel processors after one iteration.

Fig. 1, demonstrates that without load balancing, the discrepancy between the amount of work processors have to do grows with the number of iterations of the calculation. We can more acccurately characterize this growth by calculating the mean and standard deviation σ of the resulting distribution of configuration numbers. This is given in Table 1 for initial distributions fo 64, 256 and 1024 configurations per processor and for 0, 1, 2, 4, and 9 generations (steps), where generation 0 is the initial distribution. The fourth column (Percent) is the percentage that the standard deviation is of the mean. We can see that the mean remains approximately equal to the initial number of configurations per processor. Thus the total number of configurations remains the same. However the standard deviation σ grows with the number of generations. Indeed, the growth of the variance is linear in the number of steps as would be expected from a series of uncorrelated random fluctuations. This is confirmed by the fifth column (growth),

Step	σ	Mean	Percent	Growth
64 configuration per processor				
0	0.000	64.00	-	-
1	7.86	63.97	12.29	1.00
2	11.30	63.84	17.69	2.07
4	16.08	63.97	25.14	4.19
8	22.33	63.76	35.02	8.07
256 configurations per processor				
0	0.00	256.00	-	-
1	15.84	255.77	6.20	1.00
2	22.44	255.72	8.78	2.01
4	32.03	255.23	12.55	4.09
8	44.85	254.60	17.62	8.01
1024 configurations per processor				
0	0.00	1024.00	-	-
1	31.58	1023.54	3.09	1.00
2	45.32	1022.76	4.43	2.06
4	63.16	1021.84	6.18	4.00
8	90.59	1020.22	8.88	8.23

Table 1: Parameters of configuration distribution for 64 processors

which gives (σ_i/σ_1), where σ_i is the standard deviation after the i^{th} generation. Note also that the relative importance of the load imbalance due to the distribution of configurations is greater if the initial number of configurations per processor is fewer. This is confirmed by column 4, which gives the percentage that the standard deviation is of the mean. This fractional width is clearly proportional the inverse square root of the number of configurations in the ensemble, again supporting the observations that the population fluctuations are uncorrelated and random.

5.2 Elementary Load Balancing

Based on this information, our first procedure for load balancing was simply redistributing the configurations in the ensemble equally between processors. When a processor had a local population that grew large, its configurations could be given to other processors which had fewer configurations. The one additional complication that is necessary to accommodate in any load balancing scheme is:
Unbiasedness Condition *Processors can not share configurations unless they are both working on the same iteration.*
Without this condition, the Monte Carlo process does not necessarily meet the strict conditions that the final solution is unbiased and hence an exact statistical solution to the integral equation for $f(R)$, Eq. 1. Thus a synchronization requirement for sharing configurations must be imposed so that at the end of the simulation, all configuration have been iterated exactly the same number of iterations.

In our first load balancing approach, we defined a load balancing synchronization parameter, L, which is the number of iterations that the processors could carry out independently. After L iterations, a barrier was introduced and processors waited at the barrier until all other processors had completed their "L" iterations, so that the unbiasedness condition is met. At this synchronization point, an algorithm was called that selected which processors would send and receive configurations so that the number of resulting configurations would be uniform. Then, the actual send/receive operations were carried out and another L iterations were performed.

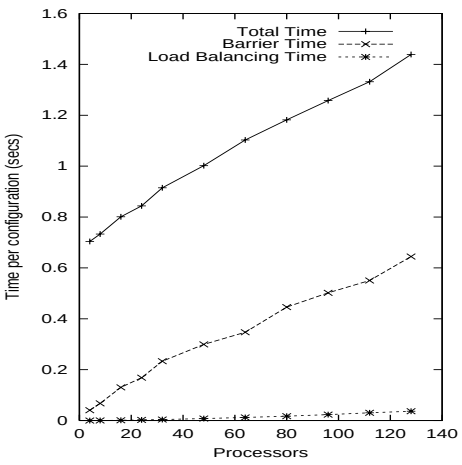

Figure 2: Time per configuration for varying number of processors.

This algorithm has an inherent inefficiency, i.e. after L iterations, synchronization of processors requires that some time is wasted. We would expect that the number of processors and the number of iterations between synchronization operations will affect efficiency. To evaluate efficiency, several performance measures were developed.

5.2.1 Performance of Elementary Load Balancing

To evaluate the performance of the elementary algorithm, we consider the standard calcularion, involving 4096 configurations for the lithium hydride calculation distributed equally over all the processors. In the initial tests, we choose $L = 4$, that is we perform load balancing every four generations.

In all the results below, useful computational work is approximately proportional to the number of configurations. Since the calculation is inherently statistical, that number fluctuates by as much as $\pm 10\%$ on different runs with the same input parameters, due to changes in the random numbers generated in the code. To remove this small fluctuation in our results, we have scaled all times to a time per configuration, i.e. CPU time or wall clock time divided by the average number of configurations that existed during the run.

Ideal parallelism would result in the total CPU time, defined as the wall clock time multiplied by the number of processors, remaining constant as the number of CPUs changed. Fig. 2 shows the dependence of the total CPU time on number of processors. We see that increasing the number of PEs by a factor of 4 from 32 to 128 changes the

total expenditure of CPU processor time per configuration from 0.9 to 1.4. In other words, the efficiency was reduced by approxiamtely 50%, although the wall clock time per configuration was reduced by a factor of 2.57.

To further understand the nature of the dependence of CPU time on load balancing and the origin of the overhead, we placed timer calls into the code that would accumulate time spent at barriers and time spend carrying out the load balancing operation itself. In addition to the exchange of configurations, this also includes the time to execute an algorithm to determine which configurations to send to which processors.

It is interesting to compare the total time per configuration in Fig. 2, to the time per configuration spent at barriers and the time per configuration spent in load balancing. Almost *none* of the increase in cumulative processor time associate with load balancing can be attributed to communications costs of the load balancing. Quantitatively, less than one per cent of the CPU time is communications time and it also represents only a few per cent of the added overhead in cumulative processor time caused by making the task parallel. On the other hand, it is clear that the increase in total time reflects very closely the increase in barrier time. Thus most of the inefficiency is caused by waiting at the barrier until all processors have reached the same generation so that load balancing can be performed in a way that does not violate unbiasedness condition. Such a large barrier time is due to the fact that the variation of the number of configurations on each processor from the initial number is significant after the first generation, as seen in Fig. 1 and Table 1.

5.2.2 Load Balancing Frequency

As shown in Table 1, when the number of steps that each processors takes gets large, the width of the distribution of the number configurations on the various processors gets larger. This suggests that the efficiency might be improved by changing the frequency of load balancing L. The number of processors used is 64, the number of configurations is 4096 and the initial values for each calculation was the same as previously. The total, barrier and load balancing time per configuration is shown in Fig. 3 for various intervals between the load balancing. We see that the total and barrier time dip is deferred until the second generation and thereafter remains constant to within the statistical variation between runs. For $L = 4$, which is the optimum valaue, the decrease is about 16%, but still this time is about 34% of total CPU time.

6 Conclusions

We considered a parallel implementation of the Green's Function Monte Carlo method for solving problems in quantum many-body physics. We characterized the randomness in processor load which arises from the nondeterministic nature of the calculation.

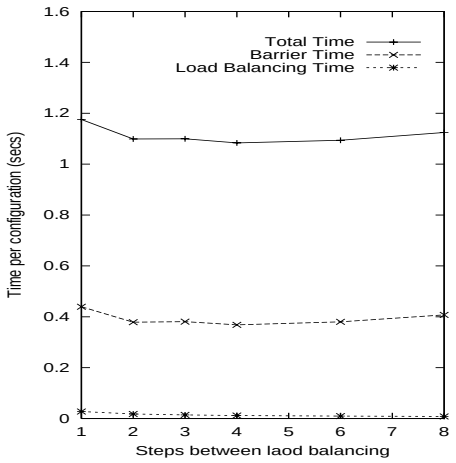

Figure 3: Time per configuration for varying numbers of steps between load balancing

We investigated an elementary load balancing algorithm and observed that even on a 128-processor run, we obtained performances which approached 50% of the theoretical maximum speed up. Thus, although considerable speedup is accomplished, it clear that scaling to hundreds or thousands of processors would be increasingly inefficient. We showed that the frequency of load balancing did not significantly effect the performance provided that that it was undertaken after two or more generations.

We further analysed the cause of this degradation in performance and showed that it was not due to communications overhead resulting from the redistribution of the configurations, but to processors having to wait at a barrier so that load distribution occurred in the same generation on each processor. This is necessary to ensure that the final solution is unbiased and hence an exact statistical solution to the integral equation. This suggests that an alternative strategy can be devised, similar to those in [2, 5], which will greatly improve the efficiency.

Acknowledgments

The authors thank Dr. Ken Flurchick of the Ohio Supercomputer Center for his intellectual contributions to algorithm development and refinement.

References

[1] M. A. Lee, K. E. Schmidt, Comp. Phys. **6**(2), 192 (1992).

[2] M. Hamdi, C. K. Lee, Parallel Comput., **22** (1997).

[3] M. H. Kalos, Phys. Rev. **128**, 1791 (1962).

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

[5] M. J. Zaki, W. Li, S. Parthasarathy, J. Parallel Distrib. Comput. **43** (1997).