

# Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code

Lorna Smith

EPCC, James Clark Maxwell Building, The King's Buildings,  
The University of Edinburgh, Mayfield Road, Edinburgh, EH9 3JZ,  
Scotland, U.K.

email: l.smith@epcc.ed.ac.uk

*Abstract*—

An OpenMP version of a Quantum Monte Carlo (QMC) code has been developed. The original parallel MPI version of the QMC code was developed by the Electronic Structure of Solids HPCI consortium in collaboration with EPCC. This code has been highly successful, and has resulted in numerous publications based on results generated on the National Cray MPP systems at EPCC. Recent interest has focussed on also utilising shared-memory parallelism in the code since future HPC systems are expected to comprise clusters of SMP nodes.

The code has been re-written to allow for an arbitrary mix of OpenMP and MPI parallelism. The various issues which arose during the parallelisation are discussed. The performance of the mixed OpenMP/MPI code has been assessed on an SGI Origin 2000 system and the results compared and contrasted to the original MPI version.

*Keywords*— OpenMP, MPI, HPC applications, performance.

## I. INTRODUCTION

The ability to study and predict theoretically the electronic properties of atoms, molecules and solids has brought about a deeper understanding of the nature and properties of real materials. The methodology used here is based on Quantum Monte Carlo (QMC) techniques, which provide an accurate description of the one-electron physics which is so important in most systems. These calculations are, however, computationally intensive and require high performance computing facilities to be able to study realistic systems.

This work has been carried out under the UK High Performance Computing Initiative (HPCI) for the highly successful Electronic Structure of Solids consortium. This consortium aims to develop of a deeper conceptual understanding of the role of quantum mechanics in determining the electronic structure and properties of real materials.

## II. BACKGROUND

Quantum Monte Carlo (QMC) methods attempt to solve the many-electron Schrödinger equation. The consortium have utilised two different QMC methods for use in their studies. The first is variational quantum Monte Carlo (VMC)[1], the second is the closely related diffusion quantum Monte Carlo (DMC) method[2]. Application of both these methods to solids use a finite simulation cell with

periodic boundary conditions to reduce the number of particles to a manageable size.

### A. Variational QMC

According to quantum mechanics, the probability that a measurement of the positions of all  $N$  electrons in a solid finds them at  $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N$  is proportional to  $|\Psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)|^2$ , where  $\Psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$  is the many-electron wavefunction. Variational QMC simulations generate sets of random positions distributed in exactly the same way as the results of this idealised measurement, and average the outcomes of many simulations to obtain quantum mechanical expectation values. Given the form of the many-electron wavefunction the required samples can be generated using the well-known Metropolis algorithm, but the exact many-electron wavefunction is an unknown function of an enormously large number of variables and has to be approximated.

### B. Diffusion QMC

In a DMC calculation, a statistical representation of the wavefunction is evolved in imaginary time so that it decays into the ground state, at which time the required averages can be accumulated. Configuration space is simultaneously explored at many points by an ensemble of “walkers” which can each dynamically breed or die. Walkers follow a guided random walk, guided by an approximate wavefunction, which greatly improves the statistical efficiency of the algorithm.

## III. THE ALGORITHM

On average, the consortium spends around 60 percent of its time carrying out diffusion Monte Carlo calculations, the rest being equally split between variational Monte Carlo calculations and wave function optimisation. Hence this work has focussed on the diffusion Monte Carlo code.

The basic structure of the DMC algorithm is:

1. Initialise an ensemble of walkers distributed with an arbitrary probability distribution.
2. Update each walker in the ensemble.  
For each electron in the walker:
  - (a) Move the electron

- (b) Calculate the local energy for the new walker position and other observables of interest.
  - (c) Calculate the new weight for this walker
  - (d) Accumulate the local energy contribution for this walker
  - (e) Breed new walkers or kill the walker based on the energy
3. Once all walkers in the current generation have been updated, evaluate the new generation averages.
  4. After  $N$  generations, calculate the new block averages
  5. Iterate steps 2–4 until equilibrium is reached; then reset all cumulative averages and iterate steps 2–4 until the variance in the run average is as small as required.

An important factor which determines the accuracy of the model is the size of the ensemble of independent configurations over which one optimises. Large ensembles up to 100 000 configurations containing several hundred electrons. The CPU and memory requirements are such that the advantages of using parallel computing are considerable.

#### IV. MESSAGE-PASSING PARALLELISATION

Like most Monte Carlo methods, DMC calculations are ideally suited to massively parallel architectures. The original parallel version of the code was developed for the Cray T3D at EPCC, using MPI. The consortium has been very successful, and has produced numerous publications[3], [4], [5], [6], [7], [8], [9].

A master-slave model is used where the master delegates work to the other processors. The master processor sends work to the slave processors who complete the required work and return the results back to the master. The numerical optimisation routine runs on the master processor and the ensemble of configurations is divided out amongst the slaves. The master processor broadcasts the resulting parameters to the slaves. Each of the slaves evaluates various quantities dependent on its subset of configurations. These are returned to the master which determines new values of the optimisation parameters. The procedure is repeated until convergence.

In the VMC calculation each processor is assigned a fixed number of electron configurations each of which undergoes an independent random walk, and interprocessor communication is needed only for the final results. However DMC calculations involve the creation and annihilation of electron configurations depending on global properties of the ensemble configurations. Before each block, or set of iterations, each processor is assigned the same fixed number of electron configurations. After each block, however the number of electron configurations on each processor may change. To avoid poor load balancing, the electron configurations are redistributed between the processors after each block. This involves a number of all-to-one communications and several point-to-point send operations.

#### V. SHARED-MEMORY PARALLELISATION

Interest in developing an OpenMP version of the code have resulted from the consortium gaining access to shared

memory systems. Further, interest in mixed OpenMP/MPI codes has recently increased with the advent of clustered SMP systems. The code has thus been re-written to allow for an arbitrary mix of OpenMP and MPI parallelism.

For the DMC calculation, the time spent carrying out re-distribution of the electron configurations depends on the data set, and this may scale poorly with increasing processor number. In principle a shared memory version of the code would eliminate this problem.

The majority of the execution time is spent within the principal DMC loop, i.e. the loop over electron configurations carried out within each block. Compiler directives have been placed around this loop allowing the work to be distributed between the threads.

By carrying out the simulation at this level of coarse-grain parallelism the MPI and OpenMP versions are distributing the work in a similar manner allowing direct comparison between the two methods.

This is a mixed mode OpenMP/MPI code and, to ensure that the code is portable to systems without thread-safe MPI implementations, MPI calls are only made from within serial regions of the code. Hence the OpenMP loop parallelisation occurs beneath the MPI parallelisation. At the start of each block electron configurations are distributed evenly between the MPI processes. The work is then further distributed by the OpenMP directives, resulting in each of the loops being executed in parallel between the OpenMP threads.

#### VI. DISCUSSION

The main DMC loop contained over 150 variables, and numerous modules and subroutine calls. Within the loop two principle shared arrays are present. At the start of the loop sections of these arrays, based on the loop index, are copied to temporary private arrays. On completion these arrays are copied back to the principle shared arrays. As the number of electron configurations, and hence the size of the temporary arrays, changes with each iteration an ORDERED statement is required to ensure the arrays are copied back in the same order as the sequential version of the code. This is a potential source of poor scaling, but was unavoidable due to the dynamic nature of the algorithm.

A large number of the underlying subroutines contain DATA and SAVE statements. These are principally logical statements used to carry out a variety of initialisations the first time the routine is called. In some cases these routines are called for the first time outside the OpenMP parallel loop and create no problem. In other situations however, these initialisations occur within the parallel loop and require private copies of the variables. This necessitated replacing these statements with THREADPRIVATE COMMON blocks.

Previous difficulties were encountered with automatic arrays, and hence the code contains a number of one-time allocations of scratch space. Hence ALLOCATE statements were used in conjunction with SAVE statements. This created problems within the parallel loop, where these arrays required to be allocated on all threads. These were returned to automatic arrays and placed in THREADPRI-

VATE COMMON blocks as allocating on the stack was no longer an issue.

Finally a number of variables are declared within a module. By default these are shared, but in the majority of cases these require to be private. There was no OpenMP solution to this and the code had to be modified to declare the variables separately inside each subroutine within the module.

These solutions were not perfect as they required some modifications of the underlying Fortran code as well as the addition of directives. We expect that problems such as these will occur in many application codes.

No problems were encountered with mixing MPI and OpenMP, and results were reproducible with a range of numbers of OpenMP threads and MPI processes. A major issue was the time spent debugging the code, primarily due to the large size of the main parallel DO loop which contained 500 lines of code, 150 variables and more than 30 calls to major subroutines. Debugging proved to be a principle factor in the development time of the code. However implementing an OpenMP version of the code was considerably easier to implement than the corresponding MPI appears to have been.

DMC calculations require a large number of calls to a random number generator. The returned values depends on the seed value, which in turn depends on previous call to the generator. The random numbers will thus be affected by the number of OpenMP threads and MPI processes, and final results will vary depending on the level of parallelism. For production runs this is not an issue as this is a statistical simulation and all results are equally valid. However as the random numbers affect the number of electron configurations this will alter the total amount of computation. For benchmark and scaling runs, where we wish the computational load to be constant, the random number generator was modified to provide consistent values with varying thread and processor number.

## VII. RESULTS

The code has been run on an SGI Origin 2000 with 40 195MHz R10000 processors. Access was only available for eight processors at a time. Significant efforts were made to ensure that processes and threads were bound to distinct processors and had exclusive access.

Timing runs have been taken for a combination of OpenMP threads and MPI processes. Figure 1 shows the scaling of the code with OpenMP thread number (with 1 MPI process) and figure 2 the scaling of the code with MPI process number (with 1 OpenMP thread).

For this particular example the code scales well with increasing MPI process number. For such low processor numbers the amount of time spent redistributing the electron configurations is relatively low and creates little overhead.

The scaling of the code with OpenMP threads is reasonable, although not as good as with MPI processes. The code was run with a combination of OpenMP threads and MPI processes, to give a total of 8. Figure 3 shows the execution time for the code for different thread/process combinations. This demonstrates that as MPI process number

increases, and OpenMP thread number decreases the execution time decreases. It is clear that increasing the number of threads decreases the efficiency of the code. The reasons for this are two fold. Firstly, as the Origin 2000 has cc-NUMA architecture with physically distributed memory, data will be stored on the node that initialised it on the Origin 2000. This is not an ideal data distribution. Further effort could be placed on distributing this data more efficiently however with the changing electron configuration the ideal data distribution is not clear. On a truly shared memory machine this will not be an issue and a better performance of the OpenMP code would be expected. Secondly, for this particular example on so few processors the time spent redistributing electron configurations is relatively small and any benefit of less MPI processors and more OpenMP threads is lost.

## VIII. CONCLUSION

An OpenMP version of a large QMC application code has been developed. The original version of the code was written in MPI and the new version has been written to explicitly allow for an arbitrary mix of OpenMP and MPI parallelism. The code scales with increasing thread number. The total time to develop a working OpenMP version was significantly less than the time taken to develop the original MPI code. However the majority of the time was spent in debugging rather than implementation. This is the inverse situation to developing a message passing application.

This study has shown that given a working MPI code there is a relatively small overhead involved in converting this to a mixed MPI/OpenMP code. We are currently porting the code to comprise SMP clusters where we expect an increased performance over the pure MPI code.

## REFERENCES

- [1] McMillan, W.L., Phys Rev 138, A442 (1965); Ceperley, D., Chester, G.V., Kalos, M.H., Phys. Rev. B 16, 3801 (1977).
- [2] Ceperley, D., Kalos, M.H., Monte Carlo Methods in Statistical Physics, Binder, K., ed., Springer-Verlag, Berlin (1984); Schmidt K.E., Kalos, M.H., Applications of the Monte Carlo Method in Statistical Physics, Binder, K., ed., Springer-Verlag, Berlin (1984).
- [3] Kent, P.R.C., Hood, R.Q., Williamson, A. J., Needs, R.J., Foulkes, W.M.C., Rajagopal, G., Finite-Size Errors in Quantum Many-Body Simulations of Extended Systems, Phys. Rev. B 59, 1917-1929 (1999).
- [4] Nekovee, M., Foulkes, W.M.C., Williamson, A. J., Rajagopal, G., Needs, R.J., A Quantum Monte Carlo Approach to the Adiabatic Connection Method, Adv. Quantum Chem. 33, 189-207 (1999).
- [5] Hood, R.Q., Chou, M.Y., Williamson, A. J., Rajagopal, G., Needs, R.J., Foulkes, W.M.C., Quantum Monte Carlo Investigation of Exchange and Correlation in Silicon, Phys. Rev. Lett. 78, 3350-3353 (1997).
- [6] G. Rajagopal and R.J. Needs, An optimised Ewald method for long-ranged potentials, J. Comput. Phys. 115, 399 (1994).
- [7] G. Rajagopal, R.J. Needs, S. Kenny, W.M.C. Foulkes and A. James, Quantum Monte Carlo calculations for solids using special -points methods, Phys. Rev. Lett. 73, 1959 (1994).
- [8] G. Rajagopal, R.J. Needs, W.M.C. Foulkes, W.M.C. Foulkes, S. Kenny and A. James, Variational and diffusion Quantum Monte Carlo calculations at Non-zero Wavevectors : Theory and application to Diamond-Structure Germanium, Phys. Rev. B 51, 10591 (1995).
- [9] S. Kenny, G. Rajagopal and R.J. Needs, Relativistic corrections to atomic energies using quantum Monte Carlo calculations, Phys. Rev. A 51, 1898 (1995).

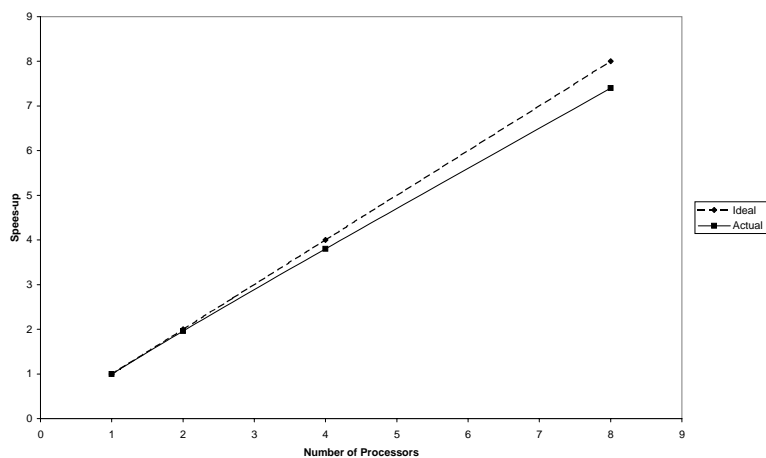


Fig. 1. Speed-up vs MPI process number on the SGI Origin 2000

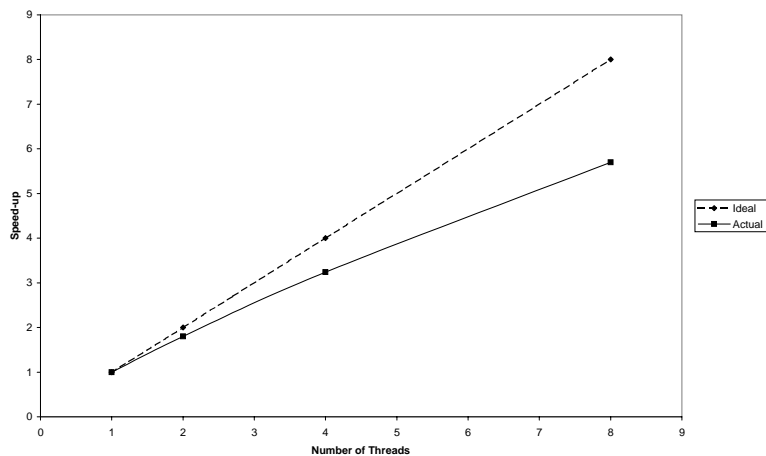


Fig. 2. Speed-up vs thread number on the SGI Origin 2000

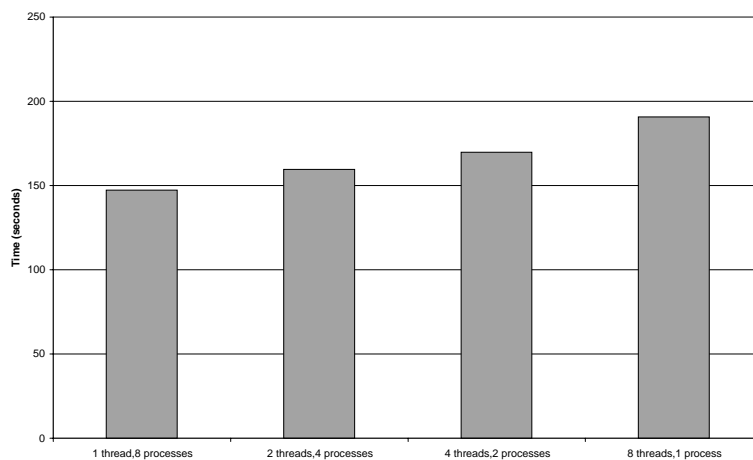


Fig. 3. Execution time for different combinations of OpenMP threads and MPI processes on the SGI Origin 2000