Development of Mixed Mode MPI / OpenMP Applications

Lorna Smith and Mark Bull

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` or `m.bull@epcc.ed.ac.uk`

Abstract—
MPI / OpenMP mixed mode codes could potentially offer the most effective parallelisation strategy for an SMP cluster, as well as allowing the different characteristics of both paradigms to be exploited to give the best performance on a single SMP. This paper discusses the implementation, development and performance of mixed mode MPI / OpenMP applications.

The results demonstrate that this style of programming will not always be the most effective mechanism on SMP systems and cannot be regarded as the ideal programming model for all codes. In some situations, however, significant benefit may be obtained from a mixed mode implementation. For example, benefit may be obtained if the parallel (MPI) code suffers from: poor scaling with MPI processes due to load imbalance or too fine a grain problem size, memory limitations due to the use of a replicated data strategy, or a restriction on the number of MPI processes combinations. In addition, if the system has a poorly optimised or limited scaling MPI implementation then a mixed mode code may increase the code performance.

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

I. Introduction

Shared memory architectures are gradually becoming more prominent in the HPC market, as advances in technology have allowed larger numbers of CPUs to have access to a single memory space. In addition, manufacturers are increasingly clustering these SMP systems together to go beyond the limits of a single system. As clustered SMPs become more prominent, it becomes more important for applications to be portable and efficient on these systems.

Message passing codes written in MPI are obviously portable and should transfer easily to clustered SMP systems. Whilst message passing is necessary to communicate *between* nodes, it is not immediately clear that this is the most efficient parallelisation technique *within* an SMP node. In theory, a shared memory model such as OpenMP should offer a more efficient parallelisation strategy within an SMP node. Hence a combination of shared memory and message passing parallelisation paradigms within the same application (mixed mode programming) may provide a more efficient parallelisation strategy than pure MPI.

Whilst mixed mode codes may involve other programming languages such as High Performance Fortran and POSIX threads, this paper will focus on mixed MPI and OpenMP codes. This is because these are likely to represent the most widespread use of mixed mode programming on SMP clusters due to their portability and the fact that they represent industry standards for distributed and shared memory systems respectively.

Whilst SMP clusters offer the greatest reason for developing a mixed mode code, both the OpenMP and MPI paradigms have different advantages and disadvantages and by developing such a model these characteristics may be exploited to give the best performance on a single SMP system.

This paper discusses the benefits of developing mixed mode MPI / OpenMP applications on both single and clustered SMPs. Section II describes related work on mixed mode programming whilst Section III provides a comparison of the different characteristics of the OpenMP and MPI paradigms. Section IV discusses the implementation of mixed mode applications and Section V describes a number of situations where mixed mode programming is potentially beneficial. Section VI contains a case study and Section VII a real mixed mode application; in both cases we describe the implementation of a mixed mode application and compare and contrast the performance of the code with pure MPI and OpenMP versions.

II. Related Work

Henty et al. [1] have developed an MPI, OpenMP and mixed mode version of a discrete element model (DEM) code and compared and contrasted the performance. This work is also discussed further in Section V. Tafti et al. [2], [3] have studied irregular applications such as adaptive mesh refinement codes which suffer from load balance problems when parallelised using MPI. By developing a mixed mode code for a clustered SMP system, MPI need only be used for communication between nodes. The OpenMP implementation does not suffer from load imbalance and hence the performance of the code would be improved. They have developed a technique called computational power balancing which dynamically adjusts the number of processors working on a particular calculation. This work is discussed further in Section V. Lanucara et al. [4] have developed mixed OpenMP / MPI versions of two Conjugate-Gradients algorithms and compared their performance to pure MPI implementations.

The DoD High Performance Computing Modernisation Program (HPCMP) Waterways Experiment Station (WES) have developed a mixed mode OpenMP / MPI version of the CGWAVE code [5]. This code is used by the US Navy for forecasting and analysis of harbour conditions. The wave components are the parameter space and each wave component creates a separate partial differential equation that is solved on the same finite element grid. MPI is used to distribute the wave components using a simple boss-worker strategy, resulting in a course grain parallelism. Each wave component results in a large sparse linear system of equations that is parallelised using OpenMP. The development of a mixed mode code has allowed these simulations to be carried out on a grid of computers, in this case on two different computers at different locations simultaneously. This mixed mode code has been very successful and won the "most effective engineering methodology" award at SC98.

Bova et al. [6] have developed mixed mode versions of five separate codes. These are the CGWAVE code mentioned above, the ab initio quantum chemistry package GAMESS, a Linear algebra study, a thin-layer Navier-Stokex solver (TLNS3D) and the seismic processing benchmark SPECseis96. Each model was developed for different reasons however most used multiple levels of parallelism, with distributed memory programming for the coarser grain parallelism and shared memory programming for the finer-grained.

Finally, a number of talks and presentations have also been given on comparing OpenMP and MPI and on mixed mode programming styles. For further information see references [7], [8], [9], [10], [11], [12], [13].

III. Programming model characteristics

As mentioned previously, as well as being potentially more effective on an SMP cluster, mixed mode programming may be of use on a single SMP, allowing the advantages of both models to be exploited. This section briefly summarises the two paradigms and considers their potential advantages and disadvantages.

A. MPI

The message passing programming model is a distributed memory model with explicit control parallelism. MPI [14] is portable to both distributed and shared memory architecture and allows static task scheduling. The explicit parallelism often provides a better performance and a number of optimised collective communication routines are available for optimal efficiency. Data placement problems are rarely observed and synchronisation occurs implicitly with subroutine calls and hence is minimised naturally.

However MPI suffers from a few deficiencies. Decomposition, development and debugging of applications can be time consuming and significant code changes are often required. Communications can create a large overhead and the code granularity often has to be large to minimise the latency. Finally, global operations can be very expensive.

B. OpenMP

OpenMP is an industry standard [15] for shared memory programming. Based on a combination of compiler directives, library routines and environment variables it is used to specify parallelism on shared memory machines. Communication is implicit and OpenMP applications are relatively easy to implement. In theory, OpenMP makes better use of the shared memory architecture. Run time scheduling is allowed and both fine and course grain parallelism are effective. OpenMP codes will however only run on shared memory machines and the placement policy of data may causes problems. Course grain parallelism often requires a parallelisation strategy similar to an MPI strategy and explicit synchronisation is required.

C. Mixed mode programming

By utilising a mixed mode programming model we should be able to take advantage of the benefits of both models. For example a mixed mode program may allow the data placement policies of MPI to be utilised with the finer grain parallelism of OpenMP. The majority of mixed mode applications involve a hierarchical model; MPI parallelisation occurring at the top level, and OpenMP parallelisation occurring below. For example, Figure 1 shows a 2D grid which has been divided geometrically between four MPI processes. These sub-arrays have then been further divided between three OpenMP threads. This model closely maps to the architecture of an SMP cluster, the MPI parallelisation occurring between the SMP nodes and the OpenMP parallelisation within the nodes.

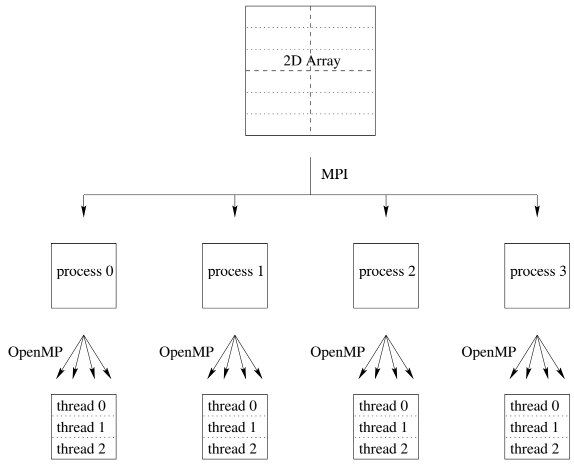

Fig. 1. Schematic representation of a hierarchical mixed mode programming model for a 2D array.

Whilst the majority of mixed mode programs implement this type of model, a number of authors have described non-hierarchical models (see Section II). For example, message passing could be used within a code when this is relatively simple to implement and shared memory parallelism used where message passing is difficult [16].

IV. Implementing a mixed mode application

Although a large number of MPI implementations are thread-safe, this cannot be guaranteed. To ensure the code is portable, all MPI calls should be made within thread sequential regions of the code. This often creates little problem as the majority of codes involve the OpenMP parallelisation occurring beneath the MPI parallelisation and hence the majority of MPI calls occur outside the OpenMP parallel regions. When MPI calls occur within an OpenMP parallel region the calls should be placed inside a CRITICAL, MASTER or SINGLE region, depending on the nature of the code. Care should be taken with SINGLE regions, as different threads may execute the code on successive passes.

Ideally, the number of threads should be set from within each MPI process using `omp_set_num_threads(n)` as this is more portable than the OMP_NUM_THREADS environment variable. Although the two models are mixed within the code, the experience of this author [17] and others [9] suggests debugging and performance optimisation is most effectively carried out by treating the MPI and OpenMP separately.

When writing a mixed mode application it is important to consider how each paradigm carries out parallelisation, and whether combining the two mechanisms provides an optimal parallelisation strategy. For example, a two dimensional grid problem may involve an MPI decomposition in one dimension and an OpenMP decomposition in one dimension. As two dimensional decomposition strategies are often more efficient than a one dimensional strategy it is important to ensure that the two decompositions occur in different dimensions.

V. Benefits of mixed mode programming

This section discusses various situations where a mixed mode code may be more efficient than a corresponding MPI implementation, whether on an SMP cluster or single SMP system.

A. *Codes which scale poorly with MPI*

One of the largest areas of potential benefit from mixed mode programming is with codes which scale poorly with increasing MPI processes. If, for example, the corresponding OpenMP version scales well then an improvement in performance may be expected for a mixed mode code. If, however, the equivalent OpenMP implementation scales poorly, it is important to consider the reasons behind the poor scaling and whether these reasons are different for the OpenMP and MPI implementations. If both versions scale poorly for different reasons, for example the MPI implementation involves too much load imbalance and the OpenMP version suffers from cache misses due to data placement problems, then a mixed version may allow the code to scale to a larger number of processors before either of these problems become apparent. If however both the MPI and OpenMP codes scale poorly for the same reason, developing a mixed mode version of the algorithms may be of little use.

A.1 Load balance problems

One of the most common reasons for an MPI code to scale poorly is load imbalance. For example irregular applications such as adaptive mesh refinement codes suffer from load balance problems when parallelised using MPI. By developing a mixed mode code for a clustered SMP system, MPI need only be used for communication between nodes, creating a coarser grained problem. The OpenMP implementation does not suffer from load imbalance and hence the performance of the code would be improved [2].

A.2 Fine grain parallelism problems

OpenMP generally gives better performance on fine grain problems, where an MPI application may become communication dominated. Hence when an application requires good scaling with a fine grain level of parallelism a mixed mode program may be more efficient. Obviously a pure OpenMP implementation would give better performance still, however on SMP clusters MPI parallelism is still required for communication between nodes. By reducing the number of MPI processes required, the scaling of the code should be improved.

For example, Henty et al. [1] have developed an MPI version of a discrete element model (DEM) code using a domain decomposition strategy and a block-cyclic distribution. In order to load balance certain problems a fine granularity is required, but this results in an increase in parallel overheads. The equivalent OpenMP implementation involves a simple block distribution of the main loop, which effectively makes the calculation load balanced. In theory therefore, the performance of a pure MPI implementation should be poorer than a pure OpenMP implementation for these fine granularity situations. A mixed mode code could provide better performance, as load balance would only be an issue between SMPs, which may be achieved with coarser granularity. This specific example is more complicated with other factors affecting the OpenMP scaling: see [1] for further details.

B. *Replicated data*

Codes written using a replicated data strategy often suffer from memory limitations and from poor scaling due to global communications. By using a mixed mode programming style on an SMP cluster, with the MPI parallelisation occurring across the nodes and the OpenMP parallelisation inside the nodes, the problem will be limited to the memory of an SMP node rather than the memory of a processor (or, to be precise, the memory of an SMP node divided by the number of processors), as is the case for a pure MPI implementation. This has obvious advantages, allowing more realistic problem sizes to be studied.

C. *Ease of implementation*

Implementing an OpenMP application is almost always regarded as simpler and quicker than implementing an MPI application. Based on this the overhead in creating a mixed mode code over an MPI code is relatively small. In general, no significant advantage in implementation time can be gained by writing a mixed mode code over an MPI code, as the MPI implementation still requires writing. There are possible exceptions where this is not the case. In a number of situations it is more efficient to carry out a parallel decomposition in multiple dimensions rather than in one, as the ratio of computation to communication increases with increasing dimension. It is however simpler to carry out a one dimensional parallel decomposition rather than a three dimensional decomposition using MPI. By writing a mixed mode version, the code would not need to scale well to as many MPI processes, as some MPI processes would be replaced by OpenMP threads. Hence, in some cases writing a mixed mode program may be easier than writing a pure MPI application, as the MPI implementation could be simpler and less scalable.

D. *Restricted MPI process applications*

A number of MPI applications require a specific number of processes to run. For example one code [18](which uses a time dependent quantum approach to scattering processes) distributes the work by assigning the tasks propagating the wavepacket at different vibrational and rotational numbers to different processes. Whilst a natural and efficient implementation, this limits the number of MPI processes to certain combinations. In addition, a large number of codes only scale to powers of 2, again limiting the number of processors. This can create a problem in two ways. Firstly the number of processes required may not equal the machine size, either being too large, making running impossible, or more commonly too small and hence making the utilisation of the machine inefficient. In addition, a number of MPP services only allow jobs of certain sizes to be run in an attempt to maximise the resource usage of the system. If the restricted number of processors does not match the size of one of the batch queues this can create real problems for running the code. By developing a mixed mode MPI/OpenMP code the natural MPI decomposition strategy can be used, running the desired number of MPI processes, and OpenMP threads used to further distribute the work, allowing all the available processes to be used effectively.

E. Poorly optimised intra-node MPI

Although a number of vendors have spent considerable amounts of time optimising their MPI implementations within a shared memory architecture, this may not always be the case. On a clustered SMP system, if the MPI implementation has not been optimised, the performance of a pure MPI application across the system may be poorer than a mixed MPI / OpenMP code. This is obviously vendor specific, but in certain cases a mixed mode code could offer significant performance improvement, for example on a Beowulf system.

F. Poor scaling of the MPI implementation

Clustered SMPs open the way for systems to be built with ever increasing numbers of processors. In certain situations the scaling of the MPI implementation itself may not match these ever increasing processor numbers or may indeed be restricted to a certain maximum number [19]. In this situation developing a mixed mode code may be of benefit (or required), as the number of MPI processes needed will be reduced and replaced with OpenMP threads.

G. Computational power balancing

A technique developed by D. Tafti and W. Huang [2], [3], computational power balancing dynamically adjusts the number of processors working on a particular calculation. The application is written as a mixed mode code with the OpenMP directives embedded under the MPI processes. Initially the work is distributed between the MPI processes, however when the load on a processor doubles the code uses the OpenMP directives to spawn a new thread on another processor. Hence when an MPI process becomes overloaded the work can be redistributed. Tafti et al. have used this technique with irregular applications such as adaptive mesh refinement (AMR) codes which suffer from load balance problems. When load imbalance occurs for an MPI application either repartition of the mesh, or mesh migration is used to improve the load balance. This is often time consuming and costly. By using computational power balancing these procedures can be avoided.

The advantages of this technique are limited by the operating policy of the system. Most systems allocate a fixed number of processors for one job and do not allow applications to grab more processors during execution. This is to ensure the most effective utilisation of the system by multiple users and it is difficult to see these policies changing. The obvious exception is "free for all" SMPs which would accommodate such a model.

VI. Case study

Having discussed the possible benefits of writing a mixed mode application, this section looks at an example code which is implemented in OpenMP, MPI and as a mixed MPI/OpenMP code.

A. The code

The code used here is a Game of Life code, a simple grid-based problem which demonstrates complex behaviour. It is a cellular automaton where the world is a 2D grid of cells which have two states, alive or dead. At each iteration the new state of the cell is entirely determined by the state of its eight nearest neighbours at the previous iteration.

The basic structure of the code is:

1. Initialise the 2D cell.
2. Carry out boundary swaps (for periodic boundary conditions).
3. Loop over the 2D grid, to determine the number of alive neighbours.
4. Up-date the 2D grid, based on the number of alive neighbours and calculate the number of alive cells.
5. Iterate steps 2–4 for the required number of iterations.
6. Write out the final 2D grid.

The majority of the computational time is spent carrying out steps 2–4.

B. Parallelisation

The aim of developing a mixed mode MPI /OpenMP code is to attempt to gain a performance improvement over a pure MPI code, allowing for the most efficient use of a cluster of SMPs. In general, this will only be achieved if a pure OpenMP version of the code gives better performance on an SMP system than a pure MPI version of the code (see Section V-A). Hence a number of pure OpenMP and MPI versions of the code have been developed and their performance compared.

B.1 MPI parallelisation

The MPI implementation involves a domain decomposition strategy. The 2D grid of cells is divided between each process, each process being responsible for updating the elements of its section of the array. Before the states of neighbouring cells can be determined copies of edge data must be swapped between neighbouring processors. Hence halo-swaps are carried out for each iteration. On completion, all the data is sent to the master process, which writes out the data. This implementation involves a number of MPI calls to create virtual topologies and derived data types, and to determine and send data to neighbouring processes. This has resulted in considerable code modification, with around 100 extra lines of code.

B.2 OpenMP parallelisation

The most natural OpenMP parallelisation strategy is to place a number of PARALLEL DO directives around the computationally intense loops of the code and an OpenMP version of the code has been implemented with PARALLEL DO directives around the outer loops of the three computationally intense components of the iterative loop (steps 2–4). This has resulted in minimal code changes, with only 15 extra lines of code.

The code has also been written using an SPMD model and the same domain decomposition strategy as the MPI code to provide a more direct comparison. The code is placed within a PARALLEL region. An extra index has been added to the main array of cells, based on the thread number. This array is shared and each thread is responsible for updating its own section of the array based on the thread index. Halo swaps are carried out between different sections of the array. Synchronisation is only required between nearest neighbour threads and, rather than force extra synchronisation between all threads using a BARRIER, a separate routine, using the FLUSH directive, has been written to carry this out.

The primary difference between this code and the MPI code is in the way in which the halo swaps are carried out. The MPI code carries out explicit message passing whilst the OpenMP code uses direct reads and writes to memory.

C. Performance

The performance of the two OpenMP codes and the MPI code has been measured with two different array sizes. Table I shows the timings of the main loop of the code (steps 2–4) and Figures 2 and 3 show the scaling of the code on array sizes 100×100 and 700×700 respectively for 10000 iterations.

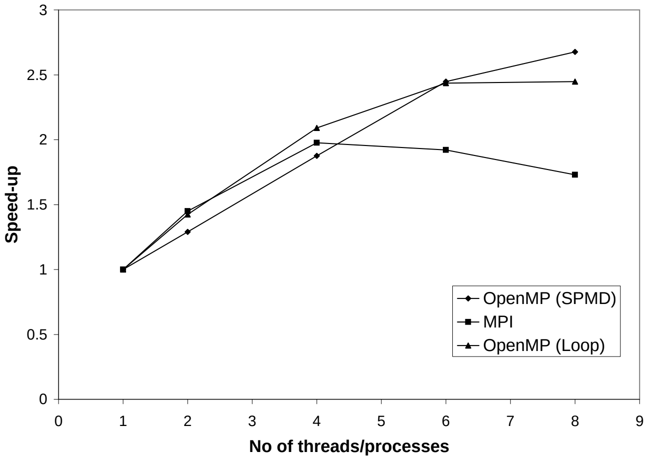

Fig. 2. Scaling of the Game of Life code for array size 100.

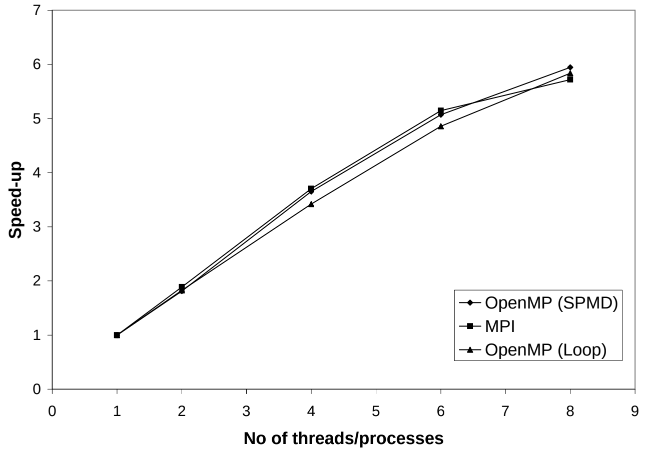

Fig. 3. Scaling of the Game of Life code for array size 700.

array size 100			
	OpenMP (SPMD)	MPI	OpenMP (Loop)
1	5.59	5.80	4.82
2	4.33	4.00	3.39
4	2.98	2.93	2.31
6	2.28	3.02	1.98
8	2.09	3.35	1.97
array size 700			
	OpenMP (SPMD)	MPI	OpenMP (Loop)
1	264.82	264.20	219.67
2	146.20	139.89	120.34
4	72.57	71.38	64.20
6	52.23	51.33	45.22
8	44.53	46.21	37.64

TABLE I

TIMINGS OF THE MAIN LOOP OF THE GAME OF LIFE CODES WITH ARRAY SIZES 100×100 AND 700×700 FOR 10000 ITERATIONS.

These results show a small difference between the timing of the OpenMP loop based code and the other codes on one processor. This is due to differences in compiler optimisation for the three codes which proved difficult to eliminate. This however has not influenced the overall conclusions of this section.

Comparison of the two OpenMP codes with the MPI implementation reveals a better performance for the OpenMP codes on both problem sizes on eight processors. It is also clear from these results that the performance difference is more extreme on the finer grain problem size. This observation concurs with Sections III-B and V-A which suggest that OpenMP implementations perform more effectively on fine grain problems. The SPMD OpenMP code gives the best overall speed-up, no matter what the problem size.

The poorer scaling of the MPI code for both problem sizes is due to the communication involved in the halo swaps, becoming more pronounced for the smaller problem size.

Both the SPMD OpenMP and MPI codes benefit from a minimum of synchronisation, which is only required between nearest neighbour threads/processes for each iteration of the loop. The loop based OpenMP implementation however involves synchronisation at the end of each PARALLEL DO region, forcing all the threads to synchronise three times within each iteration. The poorer scaling of this code in comparison to the SPMD OpenMP code is due to this added synchronisation.

These timing results demonstrate that both the OpenMP codes give better performance than the MPI code on both problem sizes. Hence developing a mixed mode MPI/OpenMP code may give better performance than the pure MPI code, and would therefore be of benefit on an SMP cluster.

D. Mixed mode parallelisation and performance

Three mixed mode versions of the code have been developed. The simplest of these involves a combination of the MPI domain decomposition parallelisation and the OpenMP loop based parallelisation. The MPI domain decomposition is carried out as before with the 2D grid divided between each process with each process responsible for updating the elements of its section of the array. Halo-swaps are carried out for each iteration. In addition OpenMP PARALLEL DO directives have been placed around the relevant loops, creating further parallelisation beneath the MPI parallelisation. Hence the work is firstly distributed by dividing the 2D grid between the MPI processes geometrically and then parallelised further using OpenMP loop based parallelisation.

The performance of this code has been measured and scaling curves determined for increasing MPI processes and OpenMP threads. The results have again been measured with two different array sizes. Table II shows the timings of the main loop of the code for 10000 iterations. Figure 4 shows the scaling of the code on array sizes 100×100 and 700×700.

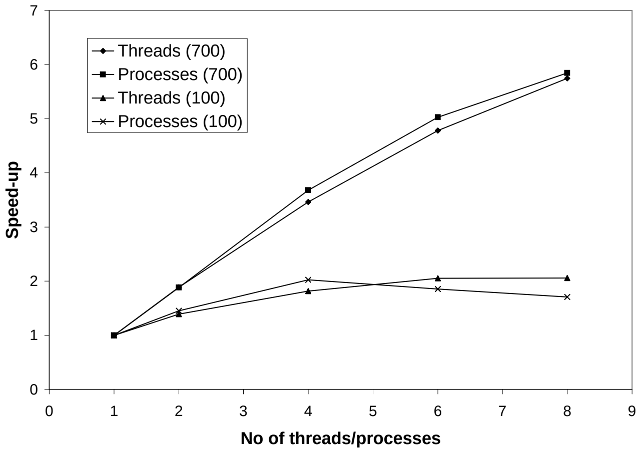

Fig. 4. Scaling of the loop based and OpenMP / MPI Game of Life code for array sizes 100 and 700.

Loop based mixed mode code				
	Threads (700)	Processes (700)	Threads (100)	Processes (100)
1	229.18	229.18	5.46	5.46
2	121.59	121.77	3.92	3.76
4	66.21	62.25	3.01	2.70
6	47.91	45.58	2.66	2.94
8	39.87	39.21	2.65	3.19
2D mixed mode code				
	Threads (700)	Processes (700)	Threads (100)	Processes (100)
1	302.41	302.41	5.22	5.22
2	172.43	161.79	3.92	3.58
4	92.69	81.77	3.86	2.52
6	68.76	59.32	3.52	2.80
8	58.51	50.00	3.56	3.09
SPMD mixed mode code				
	Threads (700)	Processes (700)	Threads (100)	Processes (100)
1	251.84	251.84	5.25	5.25
2	135.42	133.18	3.79	3.59
4	68.22	68.92	2.00	2.61
6	48.37	49.87	1.59	2.73
8	39.63	43.11	1.60	3.19

TABLE II

TIMINGS OF THE MAIN LOOP OF THE MIXED MODE GAME OF LIFE CODES WITH ARRAY SIZES 100×100 AND 700×700 FOR 10000 ITERATIONS.

It is clear that the scaling with OpenMP threads is similar to the scaling with MPI processes, for the larger problem size. However when compared to the performance of the pure MPI code the speed-up is very similar and no significant advantage has been obtained over the pure MPI implementation.

The scaling is slightly better for the smaller problem size, again demonstrating OpenMP's advantage on finer grain problem sizes.

Further analysis reveals that the poor scaling of the code is due to the same reasons as the pure MPI and OpenMP codes. The scaling with MPI processes is less than ideal due to the additional time spent carrying out halo swaps and the scaling with OpenMP threads is reduced because of the additional synchronisation creating a load balance issue.

In an attempt to improve the load balance and reduce the amount of synchronisation involved the code has been modified. Rather than using OpenMP PARALLEL DO directives around the two principal OpenMP loops (the loop to determine the number of neighbours and the loop to up-date the board based on the number of neighbours) these have been placed within a parallel region and the work divided between the threads in a geometric manner. Hence, in a similar manner to the SPMD OpenMP implementation mentioned above, the 2D grid has been divided between the threads in a geometric manner and each thread is responsible for its own section of the 2D grid. The 2D arrays are still shared between the threads. This has had two effects: firstly the amount of synchronisation has been reduced, as no synchronisation is required between each of the two loops. Secondly, the parallelisation now occurs in two dimensions, whereas previously parallelisation was in one (across the outer DO loops). This could have an effect on the load balance if the problem is relatively small. Figure 5 shows the performance of this code, with scaling curves determined for increasing MPI processes and OpenMP processes.

This figure demonstrates that the scaling of the code with OpenMP threads has decreased, and the scaling with MPI processes remained similar. Further analysis reveals that the poor scaling is still in part due to the barrier synchronisation creating load imbalance. Although the number of barriers has been decreased, synchronisation between all threads is still necessary before the MPI halo swaps can occur for each iteration. When running with increasing MPI processes (and only one OpenMP thread), synchronisation only occurs between nearest neighbour processes, and not across the entire communicator. In addition, the 2D decomposition has had a detrimental effect on the performance. This may be due to the increased number of cache lines which must be read from remote processors.

In order to eliminate this extra communication, the code has been re-written so that the OpenMP parallelisation no longer occurs underneath the MPI parallelisation.

The threads and processes have each been given a global identifier and divided into a global 2D topology. From this the nearest neighbour threads / processes have been determined and the nearest neighbour (MPI) rank has been stored. The 2D grid has been divided geometrically between the threads and processes based on the 2D topology. Halo swaps are carried out for each

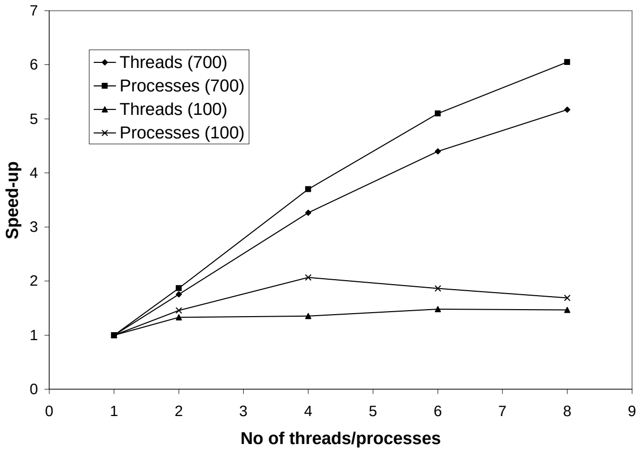

Fig. 5. Scaling of the 2D OpenMP / MPI Game of Life code for array sizes 100 and 700.

iteration of the code. If a thread/process is sending to a neighbour located on the same MPI processes (i.e. with the same rank), halo swaps are carried out using simple read and writes (as with the pure OpenMP SPMD model). If, however, the nearest neighbour is located on a different process, MPI send and receive calls are used to exchange the information. This has the effect of allowing only nearest neighbour synchronisation to be carried out, no matter how many processes or threads are available. Figure 6 shows a schematic of the halo swaps.

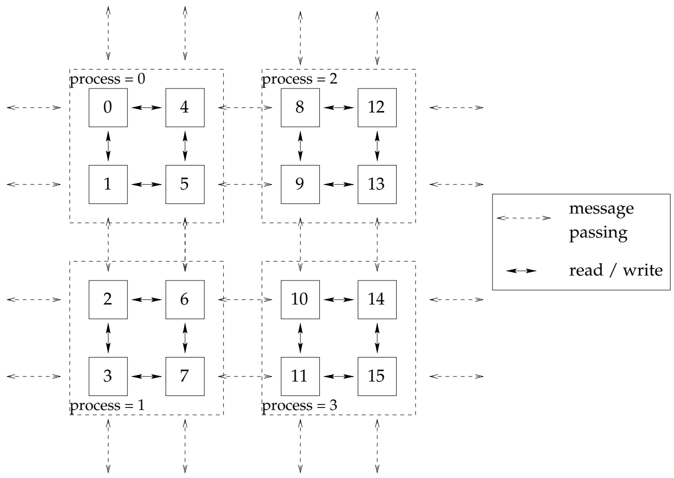

Fig. 6. Halo swaps within the mixed SPMD OpenMP / MPI code.

This does however highlight another issue. Section IV mentioned that a thread-safe MPI implementation cannot be guaranteed, and MPI calls should be carried out within thread serial regions of the code. This is still the case, however the SUN MPI 4.0 implementation being utilised has a thread safe MPI implementation and therefore allows this procedure to be carried out. Although this makes the code less portable, it allows the demonstration of the performance improvement gained by using OpenMP and MPI at the same level.

The performance of this code has been measured and scaling curves determined for increasing MPI processes and OpenMP processes. The results have again been measured with two different array sizes. Figure 7 shows the scaling of the code on array sizes 100×100 and 700×700 respectively.

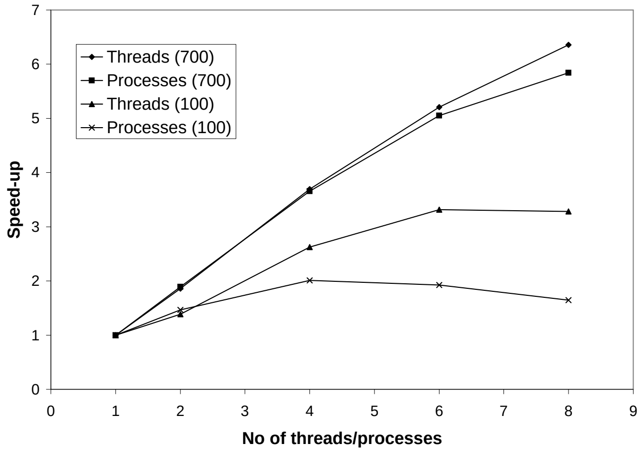

Fig. 7. Scaling of the mixed SPMD OpenMP / MPI Game of Life code for array sizes 100 and 700.

The scaling of the code with increasing OpenMP threads is greater than the scaling of the code with MPI processes. In this situation the amount of synchronisation required is the same for MPI processes as it is for OpenMP threads, i.e. only nearest neighbour. Hence the difference is only with the halo swaps, thread to thread halo swaps involving simple read and writes, whilst process to process halo swaps involving explicit message passing. Timing runs for various combinations of threads and processes, using the same total number of processors, demonstrate that as the number of threads increases and the number of processes decreases the time decreases linearly.

The aim of developing a mixed mode MPI /OpenMP code was to attempt to gain a performance improvement over a pure MPI code, allowing for the most efficient use of a cluster of SMPs. Comparison of this mixed code with the pure MPI implementation reveals a performance improvement has been obtained: the overall timings have reduced and the scaling of the code with increasing thread number is better.

E. Summary

This case study has highlighted a number of interesting points. Firstly, the OpenMP code performs better on finer grain problem sizes and therefore offers the potential for mixed MPI / OpenMP codes to give a better performance over a pure MPI implementation on these problems.

Secondly, even when a pure OpenMP implementation gives better performance over a pure MPI implementation, this does not always mean that a mixed MPI / OpenMP code will give better performance than a pure MPI implementation. For example, by implementing the mixed mode code with the MPI parallelisation above the OpenMP parallelisation, as is often the recommended case due to the lack of a guaranteed thread-safe MPI implementation, extra synchronisation is often introduced, which can reduce the performance.

Finally, for this particular example the mixed code needed to be written with MPI and OpenMP at the same level, rather than using the more common hierarchical model. This creates issues with portability, relying on a thread-safe MPI implementation, and adds to the code complexity, but increases performance.

VII. Real applications

In this section we will look at the performance of a mixed mode implementation of a real application code. This application is a mixed mode Quantum Monte-Carlo code [17]. 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 Cray MPP systems ([20], [21], [22]). Interest in developing a mixed MPI / OpenMP version of the code has recently increased with the advent of clustered SMP systems. Hence the code has been re-written to allow for an arbitrary mix of OpenMP and MPI parallelism. In this section we will briefly discuss the various issues which arose during the parallelisation and compare and contrast the performance with the original MPI version. For further details see [17].

A. *The code*

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 many-body physics which is so important in most systems. The code carries out diffusion Monte Carlo (DMC) calculations. These calculations are computationally intensive and require high performance computing facilities to be able to study realistic systems. These calculations involve a stochastic simulation where the configuration space is sampled by many points, each of which undergoes a random walk.

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 (a block), calculate the new 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 average is as small as required.

B. *MPI parallelisation*

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 master processor divides the ensemble of configurations amongst 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 parameters. The procedure is repeated until convergence.

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.

C. *Mixed mode parallelisation*

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. The principal storage arrays are recomputed before the loop over electron configurations. Although considerably less time consuming than the principle loop, this loop has an effect on the code scaling and has also been parallelised.

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. Hence the OpenMP loop parallelisation occurs beneath the MPI parallelisation.

D. *Discussion*

Within the main loop, two principal 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.

No major performance or implementation problems were encountered with mixing MPI and OpenMP, and results were reproducible with various combinations of OpenMP threads and MPI processes. As mentioned before, 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. In general the OpenMP loop parallelisation occurs beneath the MPI parallelisation. There were two exceptions to this. Firstly a number of calls to MPI_WTIME occur within the OpenMP loops, these have been modified to only allow the master thread to call them. Secondly, within the first OpenMP loop a number of MPI_BCASTs are carried out. In the original MPI code, a number of dynamically allocatable arrays are declared within a module. These are allocated the first time the routine is called, then written to once on the master process. The master processes then broadcasts the values to all the other processes. In the threaded code, OpenMP makes these arrays shared by default, but since they are written to within a parallel region they require to be private. Hence they have been returned to statically allocated arrays and placed in THREADPRIVATE COMMON blocks. All the private thread copies are then written to on the master processes. The MPI_BCASTS have been placed inside a CRITICAL section, to cause the MPI calls to only occur within a serial region while ensuring every thread on every process has a copy of the data.

E. Results

The code has been run on an SGI Origin 2000 with 300MHz R12000 processors, with exclusive access. Timing runs have been taken for a combination of OpenMP threads and MPI processes, to give a total of 96. Table III reports these timings and Figure 8 shows the scaling of the code with OpenMP thread number (with 1 MPI process) and with MPI process number (with 1 OpenMP thread).

Processes × Threads	Loop over Blocks	Processes × Threads	Loop over Blocks
1 × 1	965.64	1 × 1	965.64
1 × 4	243.52	4 × 1	241.40
1 × 8	123.25	8 × 1	120.30
1 × 16	61.98	16 × 1	60.13
1 × 32	31.54	32 × 1	30.93
1 × 64	17.17	64 × 1	15.69
1 × 96	14.65	96 × 1	11.04
1 × 96	14.65	12 × 8	11.04
2 × 48	11.11	24 × 4	11.04
4 × 24	11.05	48 × 2	11.04
8 × 12	11.04	96 × 1	11.04

TABLE III

EXECUTION TIME (SECONDS) FOR VARIOUS COMBINATIONS OF OPENMP THREADS AND MPI PROCESSES ON THE SGI ORIGIN 2000. LOOP TIMES ARE AVERAGE LOOP TIMES, AVERAGED OVER 20 ITERATIONS FOR 960 ELECTRON CONFIGURATIONS.

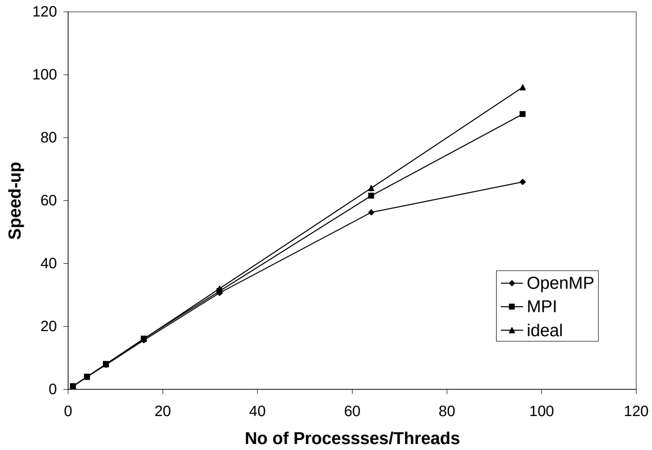

Fig. 8. Speed-up vs OpenMP thread / MPI process number on the SGI Origin 2000

For this particular example the code scales well with increasing MPI process number to 96 processors. The results also demonstrate that the scaling of the code with OpenMP threads is reasonable to 32 processors and only slightly lower than with MPI processes. Above 32 processors the scaling is worse than with MPI processes, tailing off considerably above 64 threads. Table III shows the execution time for the code for different thread/process combinations. This demonstrates that, with the exception of the 1 process × 96 threads combination, comparable results are obtained for various MPI process and OpenMP thread number.

Although the code scales well with increasing MPI process number, the scaling is not ideal. This is due to the redistribution of electron configurations between processors after each block, which involves a number of all-to-one communications and several point-to-point send operations. For example, on 96 MPI processes the redistribution of electron configurations requires 0.92s, accounting for around 8 percent of the total loop iteration time. The equivalent example on 96 OpenMP threads requires no redistribution. One possible reason for the poorer scaling of the code with OpenMP threads is the architecture of the Origin 2000. This has a cc-NUMA architecture with physically distributed memory and data will be stored on the node that initialised it. This has the potential to create a bottleneck, with all data accesses being satisfied by one hub, hence limiting the memory bandwidth. To address this problem, the data placement policy was changed to use a round-robin allocation, where data is allocated in a round-robin fashion between processors. This, however, had no effect on the scaling.

A further source of poor scaling is due to MPI calls made from within serial regions of the code. To ensure the code is portable to systems without thread-safe MPI implementations, MPI calls are only made from within serial regions of the code. As mentioned earlier, in general the OpenMP loop parallelisation occurs beneath the MPI parallelisation, with only a few exceptions. For these exceptions, the MPI calls have been placed within a CRITICAL section, to cause the MPI calls to only occur within a serial region while ensuring every thread on every process has a copy of the data. This is a potential source of poor scaling. However, the majority of these calls only occur during the first iteration of the OpenMP loop, and therefore have little effect on the performance of the code. Finally, poor scaling may be a result of the ORDERED statement within the loop over electron configurations which forces the code to be executed in the order in which iterations are executed in a sequential execution of the loop. The more dramatic tailing off above 64 processors is probably due to the ORDERED statement, which can seriously affect the performance on small problem sizes. In this case the problem size on 96 processors is relatively small and only involves 10 electron configurations per thread.

F. Summary

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 well with OpenMP threads to 32 processors and only slightly lower than with MPI processes. Above 32 processors the scaling is worse than with MPI processes, tailing off considerably above 64 threads. It is interesting to note that some of the poor scaling has been attributed to the ORDERED statement, which has effectively reduced any benefit from using OpenMP on a fine grain problem size.

VIII. CONCLUSIONS

With the increasing prominence of clustered SMPs in the HPC market, the importance of writing the most efficient and portable applications for these systems grows. Whilst message passing is required between nodes, OpenMP offers an efficient, and often considerably easier, parallelisation strategy within an SMP node. Hence a mixed mode programming model may provide the most effective strategy for an SMP cluster. In addition, a mixed mode MPI / OpenMP code has the potential to exploit the different characteristics of both paradigms to give the best performance on a single SMP.

It has, however, become clear that this style of programming will not always be the most effective mechanism on SMP systems and cannot be regarded as the ideal programming model for all codes. In practice, serious consideration must be given to the nature of the codes before embarking on a mixed mode implementation. In some situations significant benefit may be obtained from a mixed mode implementation. For example benefit may be obtained if the parallel (MPI) code suffers from:

- poor scaling with MPI processes due to e.g. load imbalance or too fine a grain problem size;
- from memory limitations due to the use of a replicated data strategy;
- from a restriction on the number of MPI process combinations.

In addition, if the system suffers from a poorly optimised or limited scaling MPI implementation then a mixed mode code may increase the code performance.

REFERENCES

[1] *Performance of hybrid message-passing and shared-memory parallelism for Discrete Element Modelling*, D.S. Henty, EPCC, The University of Edinburgh, submitted to Supercomputing 2000.

[2] *A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications*, W. Huang and D.K. Tafti, proceedings of Parallel Computational Fluid Dynamics 1999, Wiiliamsburg, VA, May 23-26, 1999.
`http://www.ncsa.uiuc.edu/SCD/Consulting/Tips/Load_Balancing.html`

[3] *Computational power balancing, Help for the overloaded processor*, D.K. Tafti.
`http://access.ncsa.uiuc.edu/Features/LoadBalancing/`

[4] *Conjugate-Gradient algortihms: An MPI-OpenMP implementation on distributed shared memory systems*, P. Lanucara and S. Rovida, proceeding of the 1st European Workshop on OpenMP, Lund, Sweden, 1999, pp76 - 78.

[5] *Dual-level parallel analysis of Harbour Wave response using MPI and OpenMP*, DoD High Performance Computing Modernisation Program (HPCMP) Waterways Experiment Station (WES).
`http://www.wes.hpc.mil/news/SC98/HPCchallenge4a.htm` and
`http://www.wes.hpc.mil/news/SC98/awardpres.pdf`.

[6] *Combining message-passing and directives in parallel applications*, S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, SIAM News, V 32, No 9.

[7] *OpenMP en plus de MPI*, Raphaël Couturier, Manifestations du Centre Charles Hermite, 1999.
`http://cch.loria.fr/activites/manifestations/1999/OpenMP_MPI/sld001.htm`.

[8] *MPI and OpenMP benchmarkers point of view*, S. Andersson, T.J. Watson Research Center presentations, IBM 1999.
`http://www.research.ibm.com/actc/Talks/StefanAndersson/MPI_OpenMP/mpi1.htm`

[9] *OpenMP programming with KAP/Pro toolset (Part 2)*, B. Magro, Kuck and Associates, Inc..
`http://www.research.ibm.com/actc/Talks/KAI/Part1/sld001.htm`

[10] *Blended programming: MPI and OpenMP*, Charles Grassel, T.J. Watson Research Center presentations, IBM 1999.
`http://www.research.ibm.com/actc/Talks/CharlesGrassl/Blended/index.htm`

[11] *Mixed-mode programming*, D. Klepacki, T.J. Watson Research Center presentations, IBM 1999.
`http://www.research.ibm.com/actc/Talks/DavidKlepacki/MixedMode/index.htm`

[12] *Experiences with mixed MPI and threaded programming models*, J.M. May and B.R. de Supinski, Center for Applied Scientific Computing, presentation at the IBM Advanced Computing Technology Center SP Scientific Applications and Optimization Meeting at the San Diego Supercomputer Center, March 18, 1999.
`http://www.llnl.gov/casc/mixed_models/pubs.html`

[13] *A comparison of OpenMP and MPI for the parallel CFD test case*, M. Resch and Bjorn Sander, proceedings of the 1St European Workshop on OpenMP, Lund, Sweden, 1999, pp71-75.

[14] *MPI*, MPI: A Message-Passing Interface standard. Message Passing Interface Forum, June 1995.
`http://www.mpi-forum.org/`.

[15] *OpenMP*, The OpenMP ARB.
`http://www.openmp.org/`.

[16] *A performance comparison of Fortran 90 with MPI and OpenMP on the Origin 2000*, J. Hoeflinger, Centre for Simulation of Advanced Rockets.
`http://polaris.cs.uiuc.edu/~hoefling/Talks/MPIvsOMP/sld001.htm`.

[17] *Development and performance of a mixed OpenMP/MPI Quantum Monte Carlo code*, L.A. Smith and P. Kent, accepted for publication in Concurrency: Practice and Experience.

[18] *Parallelism and granularity in time dependent approaches to reactive scattering calculations*, V. Piermarini, A. Laganà, G. G. Balint-Kurti, R. J. Allan.

[19] *Hybrid MPI/OpenMP programming for the SDSC teraflop system*, Scientific Computing at NPACI (SCAN).
`http://www.npaci.edu/online/v3.14/SCAN.html`.

[20] *Finite-size errors in quantum many-body simulations of extended systems*, P.R.C. Kent, R.Q. Hood, A.J. Williamson, R.J. Needs, W.M.C Foulkes, G. Rajagopal, Phys. Rev. B 59, pp 1917-1929, 1999.

[21] *A Quantum Monte Carlo approach to the adiabatic connection method*, M. Nekovee, W.M.C. Foulkes, A.J. Williamson, G. Rajagopal, R.J. Needs, Adv. Quantum Chem. 33, pp 189-207, 1999.

[22] *Quantum Monte Carlo investigation of exchange and correlation in Silicon*, R.Q. Hood, M.Y. Chou, A.J. Williamson, G. Rajagopal, R.J. Needs, W.M.C Foulkes, Phys. Rev. Lett. 78, pp 3350-3353, 1997.