Implementing Shared Memory Parallelism in MCBEND

Adam Bird[1,*], David Long[1,**], and Geoff Dobson[1,***]
[1]Amec Foster Wheeler, Kings Point House, Queen Mother Square, Poundbury, Dorchester, Dorset, DT1 3BW, United Kingdom

Abstract. MCBEND is a general purpose radiation transport Monte Carlo code from AMEC Foster Wheelers's ANSWERS® Software Service. MCBEND is well established in the UK shielding community for radiation shielding and dosimetry assessments. The existing MCBEND parallel capability effectively involves running the same calculation on many processors. This works very well except when the memory requirements of a model restrict the number of instances of a calculation that will fit on a machine. To more effectively utilise parallel hardware OpenMP has been used to implement shared memory parallelism in MCBEND. This paper describes the reasoning behind the choice of OpenMP, notes some of the challenges of multi-threading an established code such as MCBEND and assesses the performance of the parallel method implemented in MCBEND.

1 Introduction

MCBEND[1] is a well-established powerful Monte Carlo software tool for general radiation transport analysis for shielding and dosimetry applications. MCBEND is developed and licensed for use by AMEC Foster Wheeler's ANSWERS Software Service. The MCBEND package comprises not only the Monte Carlo code itself but also nuclear data libraries, user documentation, productivity tools of various kinds and user support services. Supporting geometry model visualisation and verification tools are also available.

The existing parallel capability in MCBEND known as the 'grid' option, effectively involves running the same calculation on many processors and combining the results. Because there is minimal communication required between processes the method scales almost linearly. MCBEND performs the combining of results and the user is presented with the output as if from a single run. When the accompanying user interface VisualWorkshop[2] is used the user is largely unaware of the activity. This system works well except when the memory requirements of the model are such that it reduces the number of instances that will fit on a workstation or node of a cluster.

To more effectively make use of parallel hardware the decision was taken to implement multi-threading in MCBEND in order to maximise the potential for shared-memory, and eliminating the memory constraints of the existing `grid' option. Multi-threading has been achieved within MCBEND using OpenMP[3]. The rationale for implementing a shared memory model using OpenMP is given in Section 1.1.

A description of the major design considerations and choices when implementing OpenMP are given in Section 2. Here, specific attention is given to the shared memory data management within a MCBEND calculation. Further implementation details are also given for the accumulation of scored quantities (Section 2.1) and the necessary requirement of re-producible results (Section 2.2) through the design of a new random number generator. Finally, the functional specification for the OpenMP multi-threaded parallel version of MCBEND is given in Section 2.3

1.1 Why OpenMP

When implementing a shared memory multi-threaded model there are several existing application technologies from which to choose. The following list gives the rationale behind the choice of OpenMP for MCBEND

- OpenMP is a mature technology with detailed documentation and advice on its implementation.
- OpenMP is implemented by the compiler and additional software is not required by clients to run on a given system.
- The multi-threaded behaviour is controlled by directives in comment lines within the Fortran source code. Since the implementation of OpenMP is applied on at coarse gain level over the main sample loop of a MCBEND calculation, see Section 2, load balancing considerations do not apply. Therefore, relatively few

* e-mail: Adam.Bird@amecfw.com
** e-mail: David.Long@amecfw.com
*** e-mail: Geoff.Dobson@amecfw.com

directives are required, making the implementation of OpenMP less intrusive on the existing code base and making future maintenance an easier task.

- OpenMP directives can be turned on or off, meaning that the same source code can be compiled as a sequential or parallel application. This is particularly useful when having a code base shared with other ANSWERS products and for debugging and maintenance purposes.

2 OpenMP Implementation.

In a simplified form a MCBEND calculation can be viewed in three stages:

Stage 1: Reading and set up, input data is read, data libraries are read and the data are pre-processed ready for the calculation to begin.

Stage 2: The calculation, basically a loop processing where individual particle histories are tracked from source through interaction events with the materials in the model until it escapes via absorption or moves out of the geometric or energy domain of the problem. During this processing, scores (tallies) are accumulated. The process is repeated until a specified number of histories are reached or a time limit is exceeded.

Stage 3: Post processing, where scored data are analysed and formatted for writing to output files.

Typically the reading, set up, post-processing and output, performed in stages 1 and 3, constitute <1% of the total run time for a MCBEND calculation. As such, there is very little to gain from the significant amount of effort needed to apply multi-threading to stages 1 and 3. Therefore, the multi-threaded parallelisation of MCBEND has only been applied to stage 2, the main calculation. The basis of this implementation can be seen by the simple 'fork join' model detailed in figure 1.

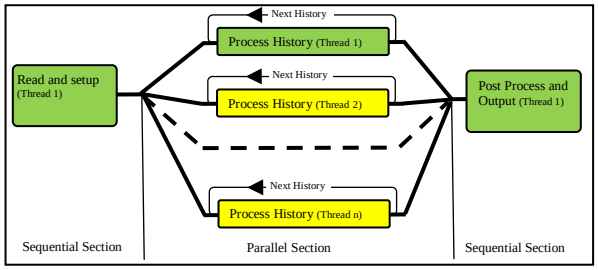

Figure 1. Multi-threading model for MCBEND.

Since each 'sample history' within stage 2 of a MCBEND shielding calculation is statistically independent from all the others, implementing multi-threading to the simple model presented in figure 1 is conceptually a simple problem involving a single coarse grain parallel region which encompasses the main sampling loop. A finer grain approach, that is using OpenMP to parallelise local sections of the code, for example, loops, was considered but it was decided that it would likely not deliver the performance or reduction of memory requirements desired.

When designing and implementing the coarse gain multi-threaded region within MCBEND, the most important consideration was for data to be correctly managed. The data within stage 2 of a MCBEND calculation fall into the following three categories.

Thread-private: Data which is private to a specific thread and should not be accessed, read from or write to, by any other thread running in the parallel region. Examples are the current attributes of individual sample histories such as the location, direction and energy.

Global read: Data which all threads within the parallel region can read, such as material cross-section values and data structures describing the geometry of the problem.

Global read-write: Data which all threads within the parallel region can read from and write to, such as scoring tally accumulators which are updated within the main sample loop when necessary.

With regards to data management it is a requirement that all thread-private data associated with a given sample should not be corrupted by other threads within the parallel region. Also, as global data cannot be specified within OpenMP as read-only, attention must be given to avoid the potential for so called 'data race' conditions when multiple threads within the parallel region have read and write access to the same variable. Violation of these conditions can lead to incorrect or irreproducible calculation results (reproducibility is necessary for verification, see Section 3).

Within the parallel region all thread-private data is duplicated for each individual thread. It was therefore critical to intelligently analyse the data present within the parallel region to identify a minimum set that needs to be thread-private. Having unnecessary amounts of thread-private data increases both the processing overhead of a multi-threaded application and the amount of memory used, which, in the limit that everything is declared thread-private, would mean that a multi-threaded MCBEND would have no advantage over the current 'grid' option.

The main effort in implementing multi-threading within MCBEND was to determine which data should be thread-private, shared globally but read only, or shared globally and writeable by all threads. In order to feasibly implement the correct assignment of all data within the parallel region, parts of the code and data structures were re-factored. The most challenging aspect of this process was to ensure that Fortran derived data types requiring to be thread-private were correctly initialised within the parallel region, as the OpenMP standard does not 'deep copy' such data structures, and therefore these had to be achieved explicitly.

Aside from the complexities of data management, the implementation of OpenMP within MCBEND has the following basic structure.

- Before entering the main parallel region, a global shared copy is made of all derived data type structures which are required to be thread-private.
- The main parallel region is opened.

- Thread-private copies of derived data types are created for each thread using globally shared initial values.
- The number of samples within the calculation is divided across the number of active threads using an OpenMP parallel 'DO' loop.
- Individual samples are tracked in parallel until all histories have been completed, with contributions from each sample to scoring tallies accumulated where necessary (see Section 2.1 for further details).
- Once tracking has been completed and the parallel 'DO' loop has been exited, any diagnostic data which has been accumulated within each individual thread is combined.
- The main parallel region is closed.
- Output files are generated sequentially.

In addition to the data management and basic structure described above, the final design of the OpenMP implementation required consideration of two remaining issues; the appropriate approach for accumulating scored quantities within the parallel region and the design of a new random number generator which generates reproducible calculation results.

2.1 Scoring Tallies

During the main calculation it is necessary to regularly update values from each sample to scoring tallies within the parallel region. For the OpenMP implementation, scoring is restricted to the Unified Tally (UT) feature of MCBEND, see Section 2.3. The use of UT scoring meshes can impose large memory usage on a MCBEND calculation, specifically when multiple fine resolution meshes are used. It is therefore impractical to define the data structures which define the UT scoring meshes as thread-private since large amounts of memory would be copied for each thread used.

Instead it was decided that the data structures responsible for storing accumulated scoring results would be shared globally, with each thread in the parallel region having write access. In order to avoid 'data race' conditions and ensure the validity of scored results, only one thread at a time is allowed to update values within a scoring mesh, achieved using the OpenMP 'ATOMIC' directive. When multiple threads attempt to synchronously update the same scoring mesh then any number of threads above unity are forced to wait idle (blocked) until the current thread has finished updating.

While using shared global data for UT scoring meshes does not require duplicating large amounts of memory, it has the potential to limit the performance of the code when increasing the number of processors used. When large numbers of threads are active in the parallel region, the likelihood that threads are blocked while updating scoring meshes is increased. This can result in a reduction of the performance gained from using additional processors, i.e. non-linear scaling.

While the issue of having shared global UT scoring mesh data potentially reducing the scaling performance is a genuine concern, it is overridden by the practical limitations duplicating thread-private UT meshes has on the memory requirements of typical system architectures used to perform MCBEND calculations. Since the maximum number of active threads for typical systems running MCBEND calculations is currently of the order of 10's, scaling issues relating to scoring within UT meshes are not expected to be fundamentally limiting to the performance of calculations. Looking to the future, as numbers of available cores increases further thought may need to be given to the treatment or scoring meshes.

2.2 Random Number Generator

The Monte-Carlo method of a MCBEND calculation uses random numbers to decide the outcome of every event that occurs within a sample history. In the sequential version of MCBEND each random number is provided by a combined lagged Fibonacci generator and linear congruential random number generator (RNG). The RNG provides a sequence of pseudo-random numbers which can be based on an initial user provided seed. If identical seeds are used for the same calculation then the same random number sequence will be generated enabling calculations to be exactly re-produced.

Unlike the sequential version of MCBEND, within the main sampling loop of the multi-threaded parallel region the exact sequence of samples is undetermined and will be different each time the same calculation is performed. Here, the n^{th} random number using the sequential RNG's will be used for a different purpose, creating different results, making it impossible to re-produce a calculation exactly.

In order to provide re-producible calculation results a new RNG has been designed for use within the multithreaded parallel region. The new RNG is seeded by a user supplied value (if present) and the unique sample number of the history being tracked. This new 'thread-safe' RNG therefore produces a unique random sequence for each sample history, which is independent of the order in which samples are tracked. This allows a direct comparison between the results from a sequential calculation with those from a multi-threaded parallel calculation, or indeed multi-threaded parallel calculations generated with differing numbers of threads. The ability to generate such reproducible calculation results is essential for verification of the multi-threaded parallel version of MCBEND as detailed in Section 3. The thread-safe RNG is the default mode when performing multi-threaded parallel MCBEND calculations.

2.3 Functional Specification

The implementation of OpenMP does not cover all of the features contained within MCBEND; implementation has been limited to a subset of the more modern functional units associated with both neutron and gamma particle tracking and scoring. The main functionality omissions that will be included later are the IGES based CAD import and electron tracking. Other omissions include the DICE and multi group collision processing,

legacy source options and some of the more advanced variance reduction methods. OpenMP multithreading has been applied to the following functionality within MCBEND;

- FG Simple body geometry, including polygon surface CAD import.
- Unified Tally.
- BINGO (Neutron and Gamma-ray collision processing).
- Splitting and Roulette (variance reduction).
- Dump and Restart.
- Hole Geometry (Woodcock tracking).
- Looping.

3 Verification

When creating the multi-threaded parallel version of MCBEND, verification that the developed code is correct is of the highest importance. The approach for verification was as follows;

- An 'intermediate' MCBEND was created that used the new random number generator, detailed in Section 2.2, with no other modifications.
- Each calculation in the test set, see Section 4, was performed using a sequential version of the 'intermediate' MCBEND.
- The results from the 'intermediate' MCBEND were compared against those from a standard MCBEND run. While the exact comparison of results for each test case is not possible they were confirmed to be statistically equivalent, which provides evidence that the new thread-safe RNG is producing the same distribution (to within acceptable limits) of random numbers as the RNG in the standard MCBEND.
- The results from the parallel OpenMP MCBEND were compared with identically seeded results from the 'intermediate' MCBEND and an exact match was expected.

Using this intermediate MCBEND allows us to prove (for the cases in the test set) that the introduction of multi-threading has not changed the basic calculation. Here, identical comparisons are only possible due to the new thread-safe RNG.

4 Test Set

For verification purposes the test set comprised 43 cases from the standard MCBEND verification set. The reduced functionality of the multi-threaded MCBEND necessitated this reduced set. Six of these models were used for performance testing of the multi-threaded version of MCBEND. This subset was chosen to cover the range of functionality that has been multi-threaded, as detailed in Section 2.3, including both FG and Hole geometry features, variance reduction through splitting and importance mesh generation, scoring of flux and response within UT meshes, different source options and collision type processing for neutron, gamma and coupled cases. A brief description of each test is given below.

- **311_bingo**: An infinite slab model with a monoenergetic uniform source. Scoring of flux occurs in a user supplied importance map mesh when boundaries are crossed.
- **fuel_flask_gamma**: A real world fuel flask example for gamma-ray dose rate calculations. Includes general, nest and array part FG features. Importance mesh is generated using the Calculate option and scoring occurs for both flux and response functions.
- **s17_t002**: Test of the Unified Tally unit for a gamma collision case with a line source.
- **s17_tab001**: Uses the pipe hole geometry with neutron collision processing. Uses looping functionality to produce tabular output for both flux and response function scored in UT mesh.
- **v2_aspis_onestep**: A coupled neutron-gamma case within a complex general part geometry. Scoring is for both flux and response function in a UT mesh.
- **v2_b2_hole**: Similar to the fuel_flask_gamma test case with the inclusion of Square and Plate hole geometry features.

5 Results

Performance testing of the multi-threaded version of MCBEND was performed on both Windows and Linux platforms. For Windows the testing was run on a standalone machine which has 12 processors. For Linux the testing was run on a single node of a HPC which has 16 processors. For each platform the multi-threaded version was compiled using the Intel Fortran 2015 compiler with the appropriate OpenMP option selected.

All test cases were run with the number of active threads in the parallel region ranging from unity to the maximum allowed for the specific platform, 12 for Windows and 16 for Linux. The results from all calculations were compared to those from the 'intermediate' version of MCBEND and were found to match identically.

Performance results were obtained by measuring the total computer processing unit time spent within the parallel region and comparing this with timings over the same code region (the main calculation sampling loop) for the 'intermediate' version. All performance timings were calculated from wall clock times obtained from calls to the Fortran system_clock function

5.1 Scaling

Scaling results for the v2_aspis_onestep test case from Windows and Linux are shown in figures 2 and 3 respectively. Here the scaling factor is plotted as a function of the number of threads used in the calculation. For each plot there are two scaling factors defined as,

$$\frac{REF_{seq}}{RUN} \quad \text{and} \quad \frac{REF_{1T}}{RUN}$$

where REF_{seq} is the reference time using the sequential 'intermediate' version of the code, REF_{1T} is the calculation time from the OpenMP version using a single thread and RUN is the calculation time from the OpenMP version for the number of threads being used

From figure 2 we find that on Windows the scaling factors when using the sequential reference times follow a linear trend based on the number of threads used, however with a constant offset from the exact ('perfect') linear scaling trend shown in black. Defining an offset factor as,

$$\left(\frac{Number\ of\ Threads}{Scaling\ Factor} - 1\right) * 100$$

the offset between the OpenMP results and the exact linear trend is approximately 50%, i.e. for 6 threads the speed up factor is 4 and for 12 threads the speed up factor is 8.

The linear trend seen in the scaling factors of figure 2 are a positive sign that OpenMP has been implemented correctly, and that performance issues related to the scoring (noted in Section 2.1) are not apparent when using the order of 10 threads. However, the relatively large offset factor noted above is less positive. While an offset factor is always expected when implementing OpenMP a value of 50% is greater than expected. A discussion of reasons behind this offset factor is given in Section 6.

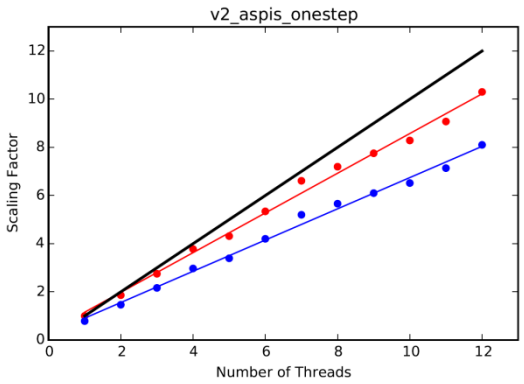

Figure 2. Scaling results for the v2_aspis_onestep test case on Windows. (blue) Scaling factor when the sequential model is used as reference run time. (red) Scaling factor when the OpenMP model with a single thread is used as the reference run time. (black) Exact linear scaling.

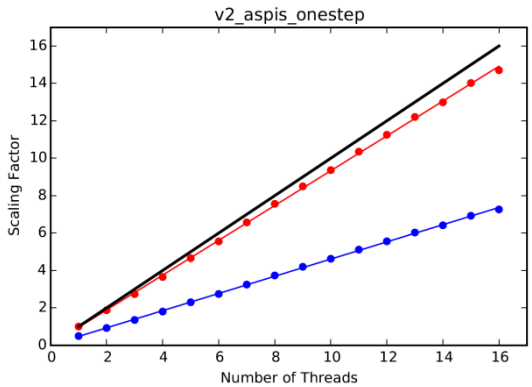

Figure 3. Scaling results for the v2_aspis_onestep test case on Linux. (blue) Scaling factor when the sequential model is used as reference run time. (red) Scaling factor when the OpenMP model with a single thread is used as the reference run time. (black) Exact linear scaling.

From figure 3 we find that the scaling results for Linux have the same form as those found for Windows. The scaling factors again follow a linear trend based on the number of threads used. However the offset factor seen when using the sequential reference times are larger than those found under Windows. Here the offset factor is of the order 120%, i.e using 8 threads results in a scaling factor of slightly less than 4 and using 16 threads results is a scaling factor of slightly less than 8.

It should be noted that all models within the test set exhibit scaling behaviour of the same form as that shown for the v2_aspis_onestep case, and as such these results are not displayed.

5.2 Test Set Performance

In addition to the scaling results presented above, the performance of all test set models under Windows and Linux are presented in tables 1 and 2 respectively. Here, SF_{SEQ} is the scaling factor when using sequential times as the reference, SF_{1T} is the scaling factor when using time from OpenMP with a single thread as the reference and the RUN times are those obtained with the maximum number of threads available on each platform,

Table 1. Windows test set performance results.

Test Name	REF_{seq}(s)	REF_{1T}(s)	RUN(s)	SF_{seq}	SF_{1T}
311_bingo	42.29	68.89	6.77	6.24	10.17
fuel_flask_gamma	94.64	119.12	12.54	7.54	9.49
S17_t002	32.51	46.69	4.6	7.06	10.14
S17_tab001	288.47	288.23	30.60	9.42	9.41
v2_aspis_onestep	100.76	128.09	12.43	8.10	10.30
v2_b2_holw	114.75	149.82	14.60	7.85	10.26

Table 2. Linux test set performance results.

Test Name	REF_{seq}(s)	REF_{1T}(s)	RUN(s)	SF_{seq}	SF_{1T}
311_bingo	14.71	35.15	2.37	6.2	14.81
fuel_flask_gamma	33.6	59.75	6.38	5.26	9.35
S17_t002	10.86	21.21	1.96	5.53	10.81
S17_tab001	100.12	180.56	19.24	5.2	9.38
v2_aspis_onestep	29.41	59.54	4.05	7.26	14.70
v2_b2_holw	39.95	70.11	7.99	4.99	8.76

From table 1 we find that the scaling factor when using the sequential reference times ranges from ~6-9.5 when using the maximum 12 threads. The offset factor for these results ranges from ~25-100% with an average offset factor of ~60%.

From table 2 we find that the scaling factor when using the sequential reference times ranges from ~5-7 when using the maximum 16 threads. The offset factor for these results ranges from ~120-200% with an average offset factor of ~180%. The performance results for the Linux platform are therefore consistently poorer than those seen under Windows.

6 Discussion and Conclusions

The magnitude of the difference in scaling factors observed when the OpenMP model with a single thread and the sequential model were used as the reference was not expected. The same source code was used to build the sequential and OpenMP versions that were used for performance testing. The only difference was the build option to enable OpenMP. What we have observed is that OpenMP running a single thread is approximately twice as slow as sequential code on Windows and between two and three times as slow on Linux.

An analysis of this performance issue using a simple test program revealed a problem associated with accessing data that is private to a thread but declared outside the scope of the parallel region, for example, Fortran module data declared using the OpenMP 'THREADPRIVATE' clause. For production code we use the Intel Fortran compiler and we have seen the problem using versions 13, 15 & 16 on the Windows and Linux operating systems. Intel has informed us that the performance issue is caused by additional run time checking that the data exists. Intel suggested not using THREADPRIVATE. This solution would require considerable re-factoring and is not a desirable option for us at this time.

Trials with version 6.1 of the Gfortran compiler on Linux do not show this problem and the difference between the OpenMP model with a single thread and the sequential case is of the order of a few percent. We are considering using an alternative compiler for the production version of the OpenMP MCBEND.

Another aspect of the performance is the memory use, especially as reducing the memory requirements of a parallel run is the main driver for this work. A multi-threaded case uses more memory than a sequential case but less than the equivalent 'grid' case. For example, a calculation using 12 threads uses four times the memory of a sequential calculation. This is a significant saving compared to the 'grid' option.

In conclusion we are very pleased with the linear nature of the scaling graphs which show, at least for the current test cases and current number of threads, that we have not yet reached a point where there is excessive interaction between threads. The main objective of reducing the memory requirements of a parallel calculation by using OpenMP has been met. The highest priority for us is to effectively remove the difference in offset factor between OpenMP with a single thread and the sequential case before we would consider a production version of MCBEND with multi-threading.

References

1. P. Cowan, G. Dobson, J. Martin, *Release of MCBEND11*, ICRS12-RPSD2012, Nara, Japan.
2. A. Bird and T. Fry, *Visual Workshop 2: A Model Viewer, Editor and Results Display Package for the Answers Shielding and Criticality Codes*, ICRS12-RPSD2012, Nara, Japan.
3. OpenMP Architecture Review Board OpenMP Application Program Interface Version 3.0, (2008).