PHYSOR 2012 – Advances in Reactor Physics – Linking Research, Industry, and Education
Knoxville, Tennessee, USA, April 15-20, 2012, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2012)

IS MONTE CARLO EMBARRASSINGLY PARALLEL?

J. Eduard Hoogenboom
Delft University of Technology
Mekelweg 15, 2629 JB Delft, The Netherlands
j.e.hoogenboom@tudelft.nl
and
Delft Nuclear Consultancy
IJsselzoom 2, 2902 LB Capelle aan den IJssel, The Netherlands
hoogenboom@delftnuclearconsultancy.nl

ABSTRACT

Monte Carlo is often stated as being embarrassingly parallel. However, running a Monte Carlo calculation, especially a reactor criticality calculation, in parallel using tens of processors shows a serious limitation in speedup and the execution time may even increase beyond a certain number of processors. In this paper the main causes of the loss of efficiency when using many processors is analyzed using a simple Monte Carlo program for criticality. The basic mechanism for parallel execution is MPI. One of the bottlenecks turn out to be the rendez-vous points in the parallel calculation used for synchronization and exchange of data between processors. This happens at least at the end of each cycle for fission source generation in order to collect the full fission source distribution for the next cycle and to estimate the effective multiplication factor, which is not only part of the requested results, but also input to the next cycle for population control. Basic improvements to overcome this limitation are suggested and tested. Also other time losses in the parallel calculation are identified.
Moreover, the threading mechanism, which allows the parallel execution of tasks based on shared memory using OpenMP, is analyzed in detail. Recommendations are given to get the maximum efficiency out of a parallel Monte Carlo calculation.

Key Words: Monte Carlo, criticality, parallel, .MPI, OpenMP

1. INTRODUCTION

The Monte Carlo method is often used as the preferred method of calculation for a nuclear reactor core because of the exact representation of almost any complex geometry and the most detailed representation of available cross section data in continuous energy. Another important feature of the Monte Carlo method is that is relatively easy to perform the calculation in parallel using multiple processors, as the particle histories to be simulated are independent of each other and can be run on different processors. This property led to the pronouncement that Monte Carlo is “embarrassingly parallel” [1,2]. Nowadays most researchers and engineers have access to clusters of computer nodes with many processors for everyday calculations. Supercomputers are available in most countries for more specialized calculations. The trend of availability of more and more processors for standard calculations will continue over the next years, if not decades. Then the parallel capabilities and efficiency of computer codes will become increasingly important and Monte Carlo methods seem to have a relative advantage over deterministic transport codes.

However, no matter how convincing the concept of Monte Carlo for parallel execution may be, in practice it turns out that the speedup of a calculation when using more processors can be severely limited. Obvious reasons are that the pre- and post-processing of the Monte Carlo calculation cannot be parallelized. Pre-processing concerns in general reading of the input, defining the geometry in the way required by the code and especially preparation of all cross section data for all nuclides present in the system. For complex problems this may take several minutes CPU time. For such complex problems it can be expected that very many particle histories need to be simulated taking much and much more time than the pre-processing, even when run in parallel. The post-processing mainly concerns collection and analysis of results and normally takes even less time than the pre-processing. So, this is not a serious threat to massively parallel applications. In some Monte Carlo codes one of the available processors for parallel execution is used as the master processor to control the activity of all other processors without doing itself any particle history simulation. With large numbers of processors this becomes less important. Hence, the cause of limited speedup in parallel execution with many processors must be found elsewhere, as will be analyzed in Sect. 2.

Parallel execution of a code on multiple processors is generally facilitated by the MPI (Message Passing Interface) standard. This standard provides a number of subroutines for Fortran or C/C++ programs to communicate between processors by sending or receiving data. Without any precautions in programming, parallel execution of a program on multiple processors will result in multiple execution of exactly the same instructions on all processors, obtaining identical outputs and results on all processors. To make multiple processors useful for a Monte Carlo code, equal parts of the total number of particle histories are executed on different processors. This still may result in identical histories on different processors if the random number generator uses the same initial seed on each processor. Hence, Monte Carlo codes will use an initial seed depending on the history number. Then truly different histories will be simulated on different processors. What remains is to collect the tally results from all processors and calculate the overall tally result.

This reveals another possible cause of inefficiency. Not only sending data from one processor to another will take time, in order to determine the overall tally result, the master processor has to wait until all processors have sent their results. Each processor can do this only after completing all the particle histories assigned to that processor. As the Monte Carlo process is by definition a stochastic process, different processors will use different times to complete their tasks and the master processor has to wait until the last processor has completed all its histories, leaving the other processors idle. Hence, not the average time per history over all histories determines the execution time, but the average time per history on the longest running processor.

2. INVESTIGATING THE EFFECTS OF PARALLEL EXECUTION

To determine the effects of parallel execution of a Monte Carlo program one performs in fact an experiment with a computer system. Like physical experiments one should be aware of various kinds of external influences on the experiment. An experimentalist will normally take measures to minimize external influences and if they cannot be excluded or made negligible, to quantify their effects on the measured quantity. It seems that this is not a problem for a computer experiments as everything seems to be under control of the programmer. We need not to monitor physical quantities like temperature, pressure, humidity, etc. They will not influence the outcome of our program. However, when measuring computer CPU or execution times, there will be external factors that do influence the result.

For measuring the CPU time a computer program uses, Fortran90 provides the intrinsic function CPU_TIME, which gives an approximation of the CPU time used. This may be satisfactory for programs run serially, but will not be useful for programs run in parallel. Then the execution time measured in terms of the difference of the wall clock time between beginning and end of the program is more useful. In fact this is the time we want to know, as it gives the time we have to wait until completion, irrespective of the fact that the CPU may not always have been working on our program in timesharing. As our computer will do more things than only running our program, the measured time is not uniquely defined and may dependent on other processes running on the computer. This even applies to a private laptop where we can make sure that only one program is started. Even on such a private laptop a lot of other programs may run autonomously and thereby influence the time measurement of our program.

Experience show that repeatedly running exactly the same computer program on the same computer will result in different execution time (or CPU time). The variation may be several per cent. As it is not easy, if not impossible, to find out the causes of the variation, it is better to accept the existence of external influences, making the measured execution time a quantity with an unknown error. It seems natural to consider a measured execution time as a random quantity with a certain standard deviation or statistical error, which can be estimated by repeating the computer experiment. However, one should be aware of the possibility that apart from the "usual" random influences on the measured execution time, it may happen that on top of that incidentally an additional error cause occurs, normally only causing a prolongation of the execution time. This may happen, for instance, when reading or writing large files while other programs on the computer also access the hard disk. This should make us reluctant in drawing conclusions from small differences in execution times. Moreover, it will be useful to repeat a computer experience, especially if there is a suspicion of irregular results for the measured execution time. An example will be shown below and one in Sect. 4. Using computer clusters for parallel execution of a program one will be assigned a number of specific processors, which cannot be used by other users of the computer system. As with the case of a laptop for private use only, this will not protect us from external influences on the execution time of our program. A demonstration is shown in Fig. 1 for an actual Monte Carlo criticality calculation. Using the standard output from MCNP, the wall clock times given in the output at the rendez-vous point after each cycle, the calculation time per cycle can be plotted. The results demonstrate not only that the calculation time per cycle can vary considerably (much more than can be expected from the random nature of particle histories), it also shows that the execution time per cycle roughly between cycles 1800 and 4600 is systematically higher than at other cycles. This cannot be caused by the intrinsic Monte Carlo calculation and must be due to other activities at the computer node or computer cluster. In general it will be difficult to detect such deviations.

Considering the effect of parallel execution of a program on multiple processors, computer architect Gene Amdahl formulated the speedup obtained with parallel processing in a mathematical way, known as Amdahl's law [3]. The speedup factor SU is determined by the number N of processors used and the fraction F of the work that can be parallelized as follows

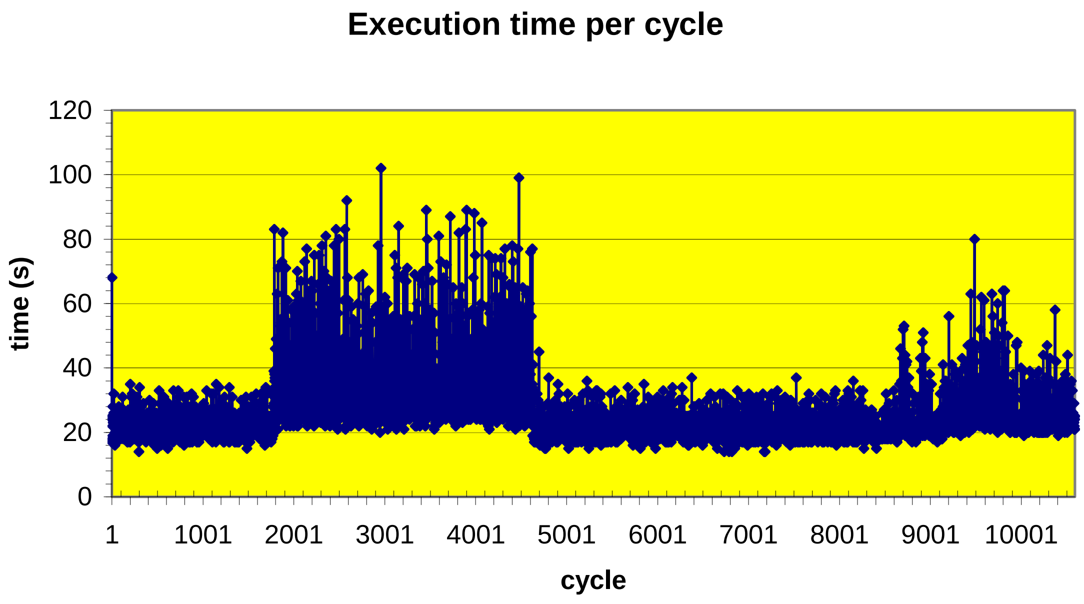

Figure 1. Execution time per cycle for a Monte Carlo criticality calculation with MCNP.

$$SU = \frac{1}{1 - F + \frac{F}{N}} \tag{1}$$

If *F*=1 the speed up factor equals the number of processors *N*. Fig. 2 shows the speedup according to Amdahl's law as a function of the number of processors for various parallelizable fractions *F* close to one. From this figure it is clear that only with fraction very close to unity one can really profit from the speedup when using tens or hundreds of processors. Therefore, it is of ultimate importance to push the parallelizable fraction to the limit when using massively parallel Monte Carlo calculations.

However, Amdahl's law does not take into account all effects than can appear in a parallel calculation and deviations from Amdahl's law can happen at larger numbers of processors. Fig. 3 shows a typical example of the effects of a parallel Monte Carlo calculation using a varying number of processors [4]. As this calculation was done with MCNP5, the master processor only distributes and collects data and does not do any neutron history simulation. This gives the shift of one processor in the graph. Already at a small number of processors the speedup stays behind the theoretically possible speedup, even if the shift of one processor is taken into account. The most important point is that when using more than about 25 processors the speedup decreases, which means that a calculation takes longer time than with fewer processors. This phenomenon is generally known and is called parallel slowdown [5]. Parallel slowdown is typically the result of a communication bottleneck. As more processing nodes are added, each processing node

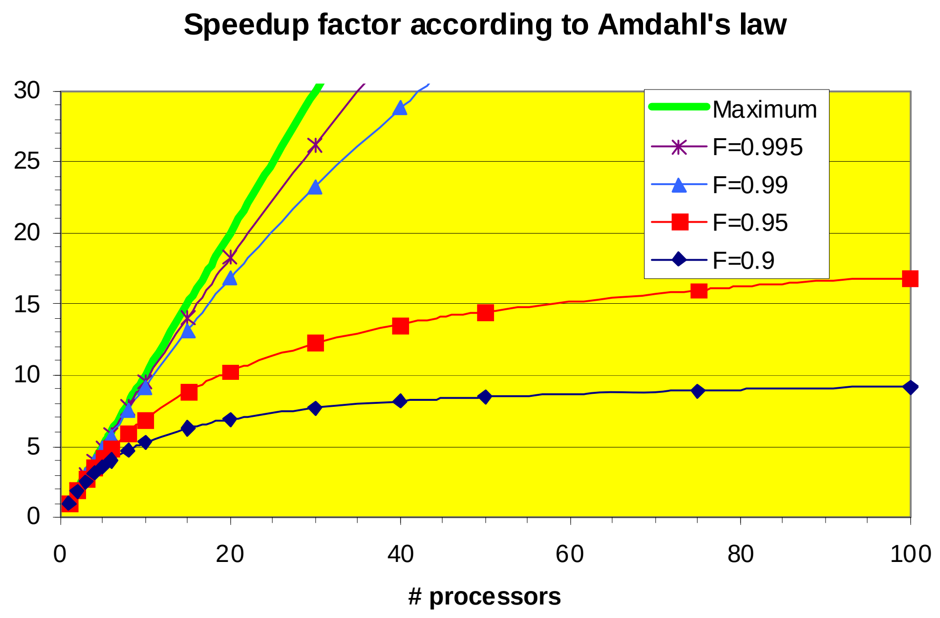

Figure 2. Speedup factor according to Amdahl’s law at increasing number of processors for various parallelizable fractions *F*.

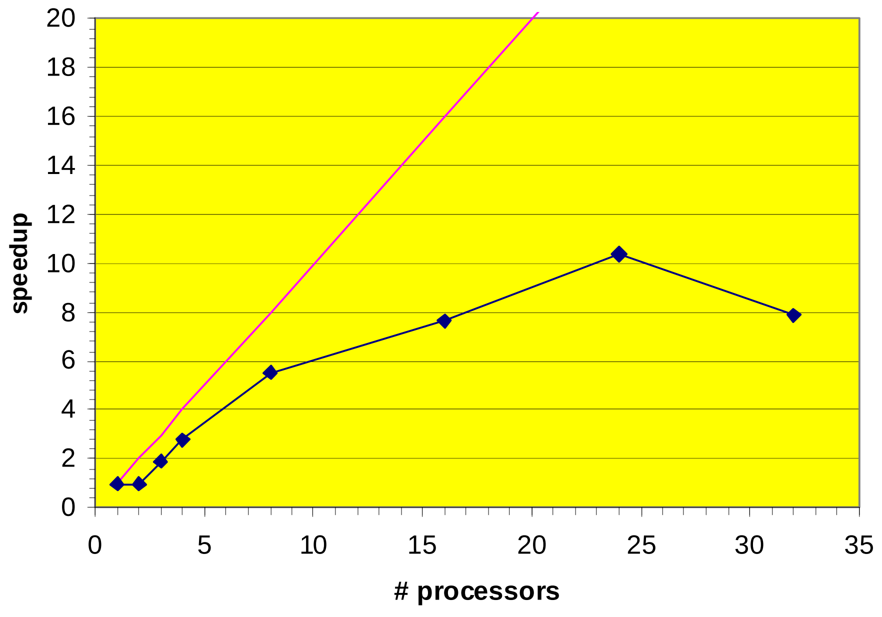

Figure 3. Example of the speedup factor at increasing number of processors for a typical Monte Carlo criticality calculation.

spends progressively more time doing communication than useful processing. At some point, the communications overhead created by adding another processing node surpasses the increased processing power that node provides, and parallel slowdown occurs.

3. PARALLEL EXECUTION ON MULTIPLE PROCESSORS WITH MPI

A Monte Carlo calculation can relatively easy be programmed to be executed on multiple processors with the use of the MPI standard [6]. This provides Fortran or C/C++ subroutines to initialize the parallel calculation, to determine the number of available processors and to identify the processor on which a copy of the program is being executed. Other subroutines can send data from one processor to another or receive that data. It is also possible to "broadcast" data from one processor (mostly the master processor) to all other processors.

For a Monte Carlo calculation the geometry and cross section data have to be sent from the master processor to all other processors, as the available memory at each processor is not shared with other processors. A criticality calculation consists of successive cycles or batches to process the neutrons from a fission source distribution and generate the fission source for the next cycle.

In our investigations we use a simple one-group criticality calculation for a slab system written in Fortan90/95. The use of a simple Monte Carlo program is acceptable for the current investigations as the complexity of actual neutron histories is not important, as long as some typical Monte Carlo characteristics are retained, like a random number of collisions for each history, resulting in randomly varying computation time per history. Apart from the source for the first cycle, the fission source obtained from a previous cycle has to be broadcasted to all processors, as well as the total number of fission neutrons and the last estimate of the effective multiplication factor for population control. Knowing the number of processors available for neutron history simulation, on each processor the range of histories numbers that should be simulated on that processor can simply be determined and the source position can be retrieved from the fission bank. Using a random number generator that is initialized with a seed depending on the neutron history number, the neutron histories over successive cycles can be reproduced independent of the number of processors used in the parallel calculation.

In our experiments we used a Linux cluster with different nodes, each consisting of 32 identical processors. A job can be submitted to the scheduling system indicating the requested number of nodes and the requested number of processors per node. As reading or writing data to a file on disk of the computer cluster can have unexpected effects on the execution time of the job, we prohibited reading or writing data to a file during the execution of the job, except at the very beginning and very end of the job. To get detailed timing information about the various steps taken in the Monte Carlo program on each processor, several calls of Fortran95 intrinsic subroutine DATE_AND_TIME, which delivers the wall clock time in hours, minutes and seconds at the moment of calling with 3 decimals for the time in seconds, providing an accuracy of 1 ms. The time is registered

- before the history simulation for a certain cycle starts on a processor
- when all histories for that cycle on that processor are completed
- when the reduce operation of the sum of tally scores and their squares is completed
- when the partial fission bank on the master processor is copied

- when the number of fission neutrons generated on each processor and the partial fission bank data are sent/received
- when the total number of fission neutrons for the next cycle and the last estimate of the effective multiplication factor are broadcasted
- when the new fission bank data are broadcasted

In order not to write the timing data to a file during the calculation, the data are stored in arrays for successive cycles on each processor and sent only once to the master processor at the end of the total calculation. The master processor then writes the timing data to a file for later analysis.

Calculations were performed, amongst others, for 50 cycles with 10^8 histories per cycle. The calculations were executed using 1, 2, 4, 8, 16 and 32 processors on the same computer node. From the total execution time $T_{exec,n}$ the speedup for a calculation with n processors was calculated by

$$SU_n = \frac{T_{exec,1}}{T_{exec,n}} \tag{2}$$

with $T_{exec,1}$ the execution time when using one processor.

Fig. 4 shows the speedup as a function of the number of processors used. Apart from the above series of calculations, also runs were executed using 4, 8 and 16 processors on two different odes each, as well as using 32 nodes on 2, 3, 4 or 5 nodes each and a few other combinations of number of processors and number of nodes. The results are also displayed in Fig. 4. Moreover, series of calculations were performed with different numbers of cycles and numbers of histories per cycle. All results for the speedup factor are displayed in Fig. 4.

From this figure several conclusions can be drawn. First, the speedup factor deviates soon from the theoretical maximum equal to the number of processors used. As the preprocessing time before the very first neutron history is started, as well as the post processing time after completion of the last cycle is small, the deviation is mainly contributed to time losses during the processing of successive cycles and collecting data from processors. Second, on our computer cluster there is a large effect in execution time whether all processors used belong to the same computer node or that processors from more than one node are used. In the last case the speedup is considerably lower and even decreases with increasing total number of processors used. This is the parallel slowdown also noted in Sect. 2. Third, the speedup factor depends on the number of cycles executed per unit execution time. Especially when using 10,000 cycles with 10^6 histories per cycle, the speedup factor is lower than in the case of 50 cycles with 10^8 histories per cycle. This is understandable as sending data more often between processors contributes more to time losses, reducing the speedup.

A further analysis using the timing data as described above for the series of runs with 50 cycles shows that the execution time for the actual neutron history simulation per processor scales well with the number of processors (the number of histories per cycle simulated on a processor is inversely proportional to the number of processors used). This time forms the major part of the total execution time for up to 32 processors. The time between the end of the neutron history simulation for a certain cycle on a certain processor and the completion of the reduce operation for the tally results is of course zero when using only one processor, but is largest when using

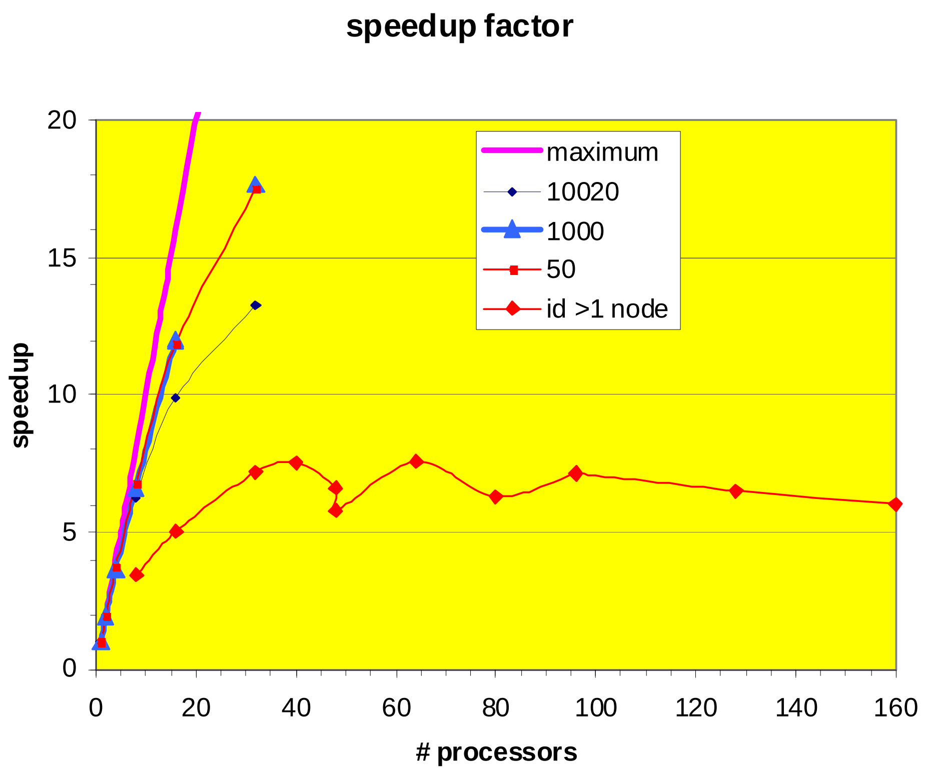

Figure 4. Speedup factor at increasing number of processors using MPI.

two processors and decreases when using more than two processors and becomes soon unimportant.

The time to copy the fission bank data on the master processor into the source array for the next cycle is proportional to the number of fission neutrons generated on the master processor and hence inversely proportional to the number of processors used. Averaged over all processors it becomes inversely proportional to the square of the number of processors and is negligible for larger numbers of processors.

The time spent for sending and receiving the fission bank data from other processors to the master processors decreases slowly from the use of two processors on. It therefore becomes a relatively important loss of time when using larger numbers of processors, as the time for actual simulation of histories per processor becomes smaller. The time spent on broadcasting the total number of fission neutrons in the next cycle and the updated value of the multiplication factor slowly increases with the use of more processors and also becomes relatively important. Finally, the time spent on broadcasting the full fission bank to all processors increases with the number of processors used and becomes the most important cause of time losses compared to the actual simulation time.

4. THREADING

Threading is the concurrent execution of multiple tasks as part of one program. Modern Fortran or C/C++ compilers will facilitate threading using the OpenMP standard [7]. This standard provides a number of compiler directives which are activated if the compilation is done with the –openmp keyword (under Unix). If this option is not activated the OpenMP directives are treated as comment lines and no threading will be applied. Threading can be applied without reservation of a number of different processors. However, for effective execution of threading more processors need to be available on the computer or computer node.

OpenMP directives are available to instruct the compiler to execute a certain part of the program in parallel on different threads. The number of threads can be set by the programmer. Different threads can share the same memory ("public" memory), which avoids copying data for geometry and cross sections as necessary for MPI. Variables that are not needed outside the threaded part of the program can be assigned memory outside the shared memory part ("private" memory).

Especially useful is the OpenMP directive to execute a given do loop in the program in parallel. The compiler will assign the execution of the loop for successive ranges of the loop variable to different threads. No further adaption of the program is needed. For a Monte Carlo calculation the obvious application is to execute the loop over successive particle histories using threading. It only requires that the random number generator is initialized with a seed depending on the particle history number, as in the case of using MPI on different processors. There are a number of limitations to the construction of the do loop to apply the OpenMP directive, but it is not the place here to go into details. There may be nested do loops within the do loop covered by the OpenMP directive. Note that in a criticality calculation there will be a do loop for successive fission source iteration cycles around the loop for particle histories within each cycle. The loop for successive cycles cannot be executed in parallel, as a certain cycle can only be executed when its preceding cycle is completely executed. However, this limitation may be surmounted by smart programming.

The use of threading is investigated with basically the same simple Monte Carlo criticality program used before for investigating MPI parallel execution. The program calculates the effective multiplication factor as the average over a fixed number of cycles for successive fission source distributions. The programming is such that it delivers the same value of k_{eff}, irrespective of the number of threads applied. As the number of threads seems basically unlimited, it is interesting to investigate how many threads can be applied in practice, how the speedup behaves when using more threads and what are the limitations.

To this end we ran the Monte Carlo criticality program on a single processor of a computer node with 32 processors. The initial number of histories is fixed at 10^6 histories per cycle and 520 successive cycles (including 20 inactive cycles, although this does not play any role in the current investigations). The number of threads is varied from 1 to 100. The only OpenMP directives used in the program are to set the number of threads, to execute the loop over the neutron histories within each cycle in parallel threads and to identify the thread number for a specific thread. In order to measure execution times per thread the wall clock time is registered using the Fortran90 intrinsic subroutine DATE_AND_TIME. The time is measured just before the threads start, after completion of the neutron history simulations on each thread and when all threads are ready. The times are averaged over all cycles of the fission source distribution. In order not to disturb the computer program with output to a file on disk, the averaged times per cycle and per thread are stored in an array, which is written to a file after completion of the run. This allows, amongst others, to estimate

- the total execution time for all fission source cycles
- the average execution time per thread (with the number of histories dependent on the number of threads used)
- the average time a thread is waiting after completion of the history simulations until all threads are ready
- the average time between the joint end of all threads and the start of new threads for the next cycle.

From the total execution time relative to the time needed for the calculation using one thread (actually a calculation without threading, which took about 1110 s) the speedup of the calculation when using a certain number of threads can be calculated. The speedup factor is shown in Fig. 5 for various numbers of threads. In the ideal case the speedup factor is equal to the number of threads used. From Fig. 5 one can see that the ideal cased is followed closely up to about 8 threads. For higher number of threads up to about 30 the speedup is still considerable. With more threads the speedup factor decreases, corresponding to a longer total execution time than with 30 threads. However, it turned out that results can strongly dependent on the local situation. For the first experiment with 10^6 neutron histories per cycle (black line) the speedup for 20 and 30 threads clearly behaves irregular. This may be due to other activities on the computer cluster during the calculation, which lasted about 1 h to repeat the experiment for 14 different numbers

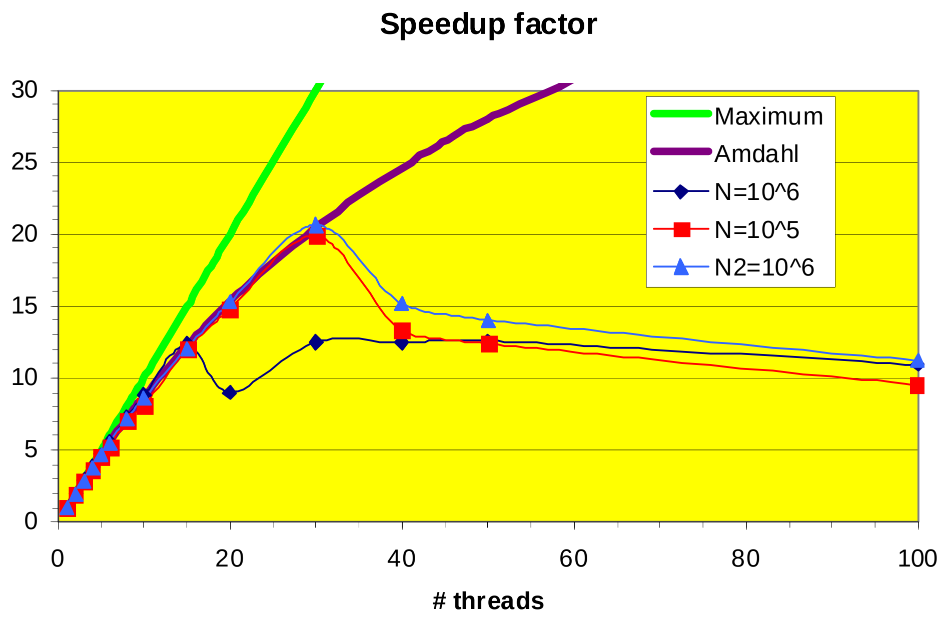

Figure 5. Speedup factor at increasing number of threads using OpenMP.

of threads. Unfortunately these conditions cannot be controlled by the experimenter. A second experiment with 10^6 neutron histories per cycle (blue line) gave more consistent results, also in line with the experiment with 10^5 neutron histories per cycle.

Analysis of the detailed data of the experiments show that up to about 15 threads the sum of actual execution times per thread closely equals the execution time for one thread. The deviation from the ideal line is then caused by the time between the completion of all threads during a cycle and the start of new threads for the next cycle, as well as the time a thread has to wait until completion of all other threads before the computer program can be continued. In our experiments with 10^6 neutron histories per cycle the average time between completion of all threads and the start of new threads shows some variation, but is on the average about 30 ms. This is the time of the non-parallelizable part. Hence the parallelizable fraction *F* in terms of Ahmdahl's law can be calculated as 1 - 0.030 x 520/1110=0.986. As the total execution time decreases when using more threads, the non-parallelizable time weighs more heavily with increasing number of threads. However, also the time a thread has to wait until all other threads are finished decreases the speedup factor. This time can vary considerably from thread to thread and from cycle to cycle, but is on the average 5 ms. Although this is strictly speaking not a non-parallelizable part, it effectively behaves as such, bringing the parallelizable fraction *F* down to 0.984. The speedup according to Amdahl's law is also shown in Fig. 5. With these numbers the total execution time for a certain number of threads can be explained. For instance, in the case of 20 threads the actual execution time for all histories in parallel is 1110/20=55.5 s. The time lost by waiting and the non-parallelizable part is 520 x (30+5) ms = 18.2 s. Hence the total execution time is estimated to be 73.7 s, while the actual execution time for this case was 72.5 s. Then the speed up factor is 1110/72.5=15.3. In case of parallel execution with MPI on multiple processors the waiting time could be reduced by load balancing. This is not possible with threading. In our case, with 32 processors in a computer node, the measured execution time per history increases considerably when using more than 32 threads as threads can no longer be executed really concurrently, which causes the decrease in speedup factor for large number of threads.

The speedup, up to about 30 threads, is limited mainly by the time used for the non-parallelizable part in between two successive cycles. In our case this time is probably mainly determined by reorganizing the fission source bank between two successive cycles. This time can be reduced by smarter programming. As the number of threads is limited anyway, this may not be that important for optimization of threading, but it is of ultimate importance for parallel execution on a computer with a very large number of processors.

The PBS scheduling system on the Linux cluster used for these investigations requires that the number of processors to be used in the job is set at scheduling time. The above calculations were run reserving one processor. However, the threads effectively use other available processors on the computer node. When other users on the computer cluster submit jobs that use one or more of the available processors on the node, the maximum number of threads that can be used without deterioration of the execution time will be limited. Hence, it is necessary from a practical point of view that a number of processors is reserved at scheduling time equal to the maximum number of threads in the Monte Carlo program.

5. COMBINING MPI AND THREADING

For parallel execution of a job the MPI and OpenMP mechanism can simply be combined as the programming of both mechanisms can be made independent. A general purpose Monte Carlo code like MCNP5 also offers this option. However, with MCNP5 it is not possible to run a job with threading only, without using MPI. In MCNP5 the last requirement forces the use of at least 3 processors for MPI, as one processor is used solely for distributing data and collecting results.

One should realize that there are severe limitations in the efficiency when combining the MPI and OpenMP mechanisms. To investigate what happens on a Linux cluster we ran a series of Monte Carlo criticality calculations reserving all 32 processors on the computer node at scheduling time, but running the parallel execution using MPI on a varying number of processors using the `mpiexec -n` command option. Fig. 6 shows some results for the speedup. The legend indicates the number of processors assigned for MPI. The speedup factor when using *n* processors with MPI and *m* threads is now defined as

$$SU_{n,m} = \frac{T_{exec,n,1}}{T_{exec,n,m}} \tag{3}$$

with $T_{exec,n,m}$ the execution time when using *n* MPI processors and *m* threads.

From Fig. 6 we can draw several important conclusions. First, the speedup factor decreases suddenly and considerably when the number of processors used for MPI times the number of threads becomes larger than the total number of available processors on the computer node. This

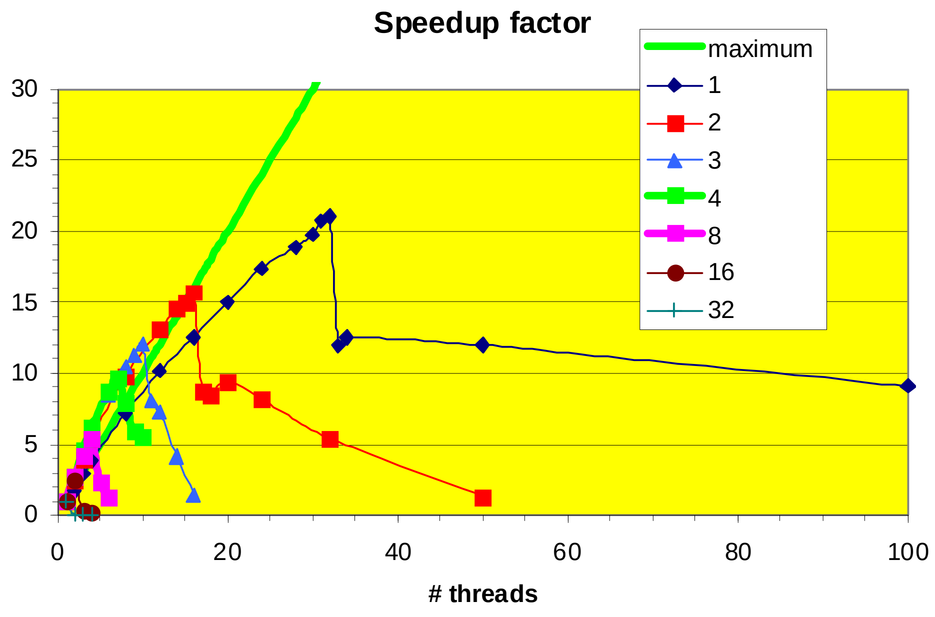

Figure 6. Speedup factor at increasing number of threads using various numbers of processors for MPI.

fact should be taking into account when determining the scheduling parameters and the number of threads. As with threading only, the total number of processors used for the job should be reserved at scheduling time to avoid that another user will occupy one or more processors on the limiting the available number of processors for the Monte Carlo program, with possible extreme increase of the actual execution time of the job. For the case with 3 MPI processors the maximum number of threads is [32/3]=10.

Second, the combination of *n* MPI processors and *m* threads is more or less equivalent with *m* MPI processors and *n* threads, provided that *n* x *m* processors are totally available. The case of 32 MPI processors with a single thread per processor took 27 % longer execution time than the case of a single processor using 32 threads. For other combinations the difference is much less. Third, effective combination of MPI and OpenMP can be advantageous, resulting in a somewhat larger speedup than maximally possible when using more threads for a fixed number of MPI processors (points at the left of the linear speedup line equal to the number of threads). From the measured execution times for all cases considered, the minimum execution time is obtained (for our case with a maximum of 32 processors available) at 8 MPI processors and 4 threads.

The above investigations refer to running the Monte Carlo job on multiple processors at the same computer node. Some experiments using processors divided over two nodes show that this seriously limit the possibilities of threading. As can be expected, the number of threads cannot be larger than the number of processors available on one node without considerably decreasing the speedup. For instance, when using 2 nodes with each 8 processors, the number of threads should be limited to 8. Using, in this case, 4 MPI processors on each node with the `mpiexec -n 4` command and 2 threads or 2 MPI nodes and 4 threads is near optimal.

6. HOW TO IMPROVE PARALLEL EXECUTION?

The above analysis of parallel execution of Monte Carlo criticality calculations results in indications how the speedup of a calculation can be improved. The main improvements must be found in reducing the time losses when using many processors.

In Sect. 3 we noted that broadcasting the fission source distribution from the master processor to all other processors will give the major loss of time when using many processors. This loss can be reduced in several ways. First, it is not needed to send the total fission source distribution of all fission neutrons to each processor, as each processor will only simulate neutron histories for a fraction of the total number of fission neutrons. Hence, the master processor can send only the relevant part of the total fission source distribution for each cycle to a certain processor. This should reduce the communication time roughly by a factor equal to the number of processors. However, as the data to be sent to a specific processor is now dependent on that processor, the broadcast mechanism can no longer be used, but a send/receive mechanism should be used instead. Other possibilities are to realize that the major part of the fission neutrons generated on a certain processor will come back to the same processor after collecting all fission source data on the master processor and redistributing them over all processors. Hence, one could limit oneself by redistributing only a small part of the fission neutron data from one processor to their direct neighbors, saving a lot of data communication between processors [8].

A more radical solution will be not to collect all fission source data by the master processor at the end of each cycle and redistribute them again in equal parts over all processors. It is possible to continue successive cycles on each processor with the fission neutrons generated on that processor. Then no data communication is needed at all between processors. Only at the very end of the calculation the tally results per processor must be collected to calculate the average over all processors and possibly the last fission source data must be collected if one wants to save these data in a file for later use. This will be the most efficient model with respect to minimization of communication between processors. This method is used, for instance, in the SERPENT and TRIPOLI4 Monte Carlo codes.

This way of handling the fission source distribution for successive cycles was implemented in our simple Monte Carlo program and the execution time measured for various numbers of processors using MPI. Table I shows the results for the speedup as in a graph the results cannot be clearly distinguished from the ideal maximum speedup. Hence, a serious improvement can be attained in this way. An additional advantage is that the negative effect on the execution time when using processors on different computer nodes is almost absent.

Table I. Speedup factor without processor communication

# processors	Execution time (s)	Speedup factor
1	7823.9	1
2	3913.0	2.00
4	1967.4	3.98
8	984.9	7.96
16	493.5	15.92
32	249.6	31.63

This method of keeping the fission source distribution of successive cycles within the processor has the disadvantage that the outcome of the Monte Carlo calculation cannot be reproduced exactly when using different numbers of processors. This can be overcome by a modest exchange of data between processors, namely the number of fission neutrons generated on each processor for the next cycle and the effective multiplication factor obtained on each processor. These data should be shared with all processors. The number of fission neutrons per processor is needed to determine the initial seed for a new history and the multiplication factors per processor must be combined to the overall multiplication factor, which is needed for population control in the next cycle. Then it will be possible to run all histories independently of the number of processors used at the cost of some communication and synchronization time. In this case it is very important to apply effective population control in order to force the number of fission neutrons in successive cycles on each processor the same as closely as possible. This way of keeping the results reproducible independent of the number of processors was also programmed and the execution times measured. The general conclusion is that the speedup only marginally decreases. Hence, it is a practical method.

7. CONCLUSIONS

In this paper a number of issues are addressed that are relevant for parallel execution of Monte Carlo calculations, especially criticality calculations. It is shown that both the MPI and OpenMP mechanism for parallel execution can speedup a calculation considerably, but that both techniques will show limitations in speedup when using large numbers of processors. The investigations concern the use of a Linux computer cluster using one or more nodes, each consisting of up to 32 processors. It was also concluded that using processors spread over 2 or more nodes may deteriorate the speedup and may result in no gain in execution time compared to a single computer node.

The relative increase in execution time when using larger numbers of processors is due to the communication time between processors (usually between each processor and the master processor) and the time processors have to wait when all their histories are processed for the master processor to have collected results from all processors. Another loss is due to the distribution of the fission source for a new batch or cycle to all processors. These factors may limit or even inhibit using massively parallel calculations on a supercomputer. However, as the computer architecture and communication channels are different from those of a Linux cluster it needs further investigations how efficient a Monte Carlo criticality calculation can be performed on a supercomputer with very many processors.

Improvements for the efficiency of parallel execution concern mainly the limitation of data exchange between processors after each cycle of fission source determination. As discussed in Sect. 5 it is possible to fully exclude data exchange after each cycle, which may become the obvious way for massively parallel Monte Carlo criticality calculations. Another factor that limits the parallel efficiency is the use files on disk to write data to during the calculations. Where possible it will be better to collect as much data in memory on each processor and to collect the data at the end of the calculation and then write it to disk.

The item of load balancing was not investigated. Load balancing can be applied if one processor has completed its histories while another processor still has to do several histories. However, from our investigations it can be concluded that the effect of load balancing will be only marginal because the difference in execution time per cycle over different processors is relatively small and it will need relatively much communication and synchronization to realize this option.

With respect to the differences between MPI and OpenMP it seems that threading with OpenMP has advantages over using MPI as it uses shared memory and more effective communication to collect results over all threads. Both methods can be combined and a certain combination may be most profitable. However, a necessary condition is that a number of processors must be reserved on a single computer node at the time of scheduling the job equal to the product of the number of MPI processors used via the `mpiexec` command and the number of threads. If this product is larger than the available number of processors, serious deterioration of the execution time will occur. Hence, one should be careful in using the combination of MPI and OpenMP.

For the investigations in this paper a very simple Monte Carlo program was used. Although this should not matter for the general conclusions, it will be necessary to investigate in more detail the behavior of general purpose Monte Carlo codes in parallel execution using

ACKNOWLEDGMENT

The research leading to these results is partly funded by the European Atomic Energy Community's (Euratom) Seventh Framework Program FP7/2007-2011 under grant agreement № 295971.

REFERENCES

1. "Embarrassingly parallel," http://en.wikipedia.org/wiki/Embarrassingly_parallel
2. R.G. Brown, *Engineering a Beowulf-style Computer Cluster* (2009), http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/node30.html
3. "Amdahl's law," http://en.wikipedia.org/wiki/Amdahl%27s_law
4. J.E. Hoogenboom, W.R. Martin and B. Petrovic, "The Monte Carlo Performance Benchmark Test – Aims, Specifications and First Results," *Proceeding of M&C2011 Conference*, Rio de Janeiro, Brazil, May 8-12, 2011 (2011).
5. "Parallel slowdown," http://en.wikipedia.org/wiki/Parallel_slowdown
6. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*, High Performance Computing Center, Stuttgart, Germany (2009); also available from http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.
7. OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1," (2011); available from http://www.openmp.org/mp-documents/OpenMP3.1.pdf.
8. P.K. Romano and B. Forget, "Fission Bank Algorithms in Monte Carlo Criticality Calculations," to be published in *Nucl. Sci. Eng.*