RAPID MONTE CARLO SIMULATIONS USING PARALLEL COMPUTING AND A CLIENT-SERVER MODEL

RÜDIGER KESSEL[†]

Fachbereich Stoffeigenschaften und Druck, Physikalisch-Technische Bundesanstalt (PTB), Bundesallee 100 D-38116 Braunschweig, Germany

RAGHU KACKER

Applied and Computational Mathematics Division, National Institute of Standards and Technology (NIST), 100 Bureau Drive, Gaithersburg, MD 20899-8910, USA

With the publication of Supplement 1 to the Guide to the Expression of Uncertainty in measurement (GUM), an alternative calculation method is available in metrology to evaluate measurement results and their uncertainties. The calculation method is based on Monte Carlo simulation techniques and it employs a large number of repeated numerical calculations. Depending on the measurement model, the computational effort can be large and time consuming. Processors with multiple CPU cores have become widely available. Therefore it is useful to implement and test concepts to parallelize the Monte Carlo simulation process in the context of metrology. The development of an Open Monte Carlo Engine (OMCE) for metrology in Python offers the possibility to implement some parallel computing concepts in practice. Different approaches for parallel computing including Monte Carlo simulations are available and have been studied. We choose an approach which is based on a client-server model and which allows the use of remote computing resources if they are available. The communication between clients and servers is socket based using a high level remote procedure call protocol which integrates different system architectures (e.g. Windows and Linux) in an overall simulation “network”. The user can access the simulation network via a lightweight client which delegates all calculations to his network entry server. The network entry server handles the job and employs other servers or CPU cores as far as they are available. The results are communicated back to the client when the simulation task is finished. The system is optimized to limit the communication between client and server to a minimum and it is buffered to allow the calculation processes to continue, even if communication might be slow. The concept is flexible enough to be used by computers with multiple CPU cores or by multiple computers connected by a fast local network (LAN) or any combination of both. The key element for the quality of the Monte Carlo simulation is the use of uncorrelated parallel random number generators in the cooperating servers. We analyze different techniques and implement an appropriate seeding method. We share some preliminary results about practical measurements of the simulation speed that can be achieved with a 48 CPU core system under Linux and we

[†] E-mail address: ruediger.kessel@ptb.de

analyze how well our concepts will scale with increasing number of CPUs included in the network.

1. Implementation of the Monte Carlo method

The Guide to the Expression of Uncertainty in measurement (GUM) [1] defines the principles for the evaluation of results in measurement. In its Supplement 1 [2] an alternative calculation based on the Monte Carlo method has been defined which employs a large number of repeated numerical calculations. These methods have been implemented in the Open Monte Carlo Engine (OMCE) [3] which is an open source general purpose simulator implemented in Python [4].

The OMCE is a command line tool (script) which uses a description of metrological model in XML-format as an input and creates text and binary output. The details of the processing are controlled by command line options.

Figure 1 shows a block diagram with the processing task structure implemented by the OMCE. The OMCE uses multi-tasking to interleave calculations and binary data save, but cannot make use of multiple processing cores because of the Python global interpreter lock.

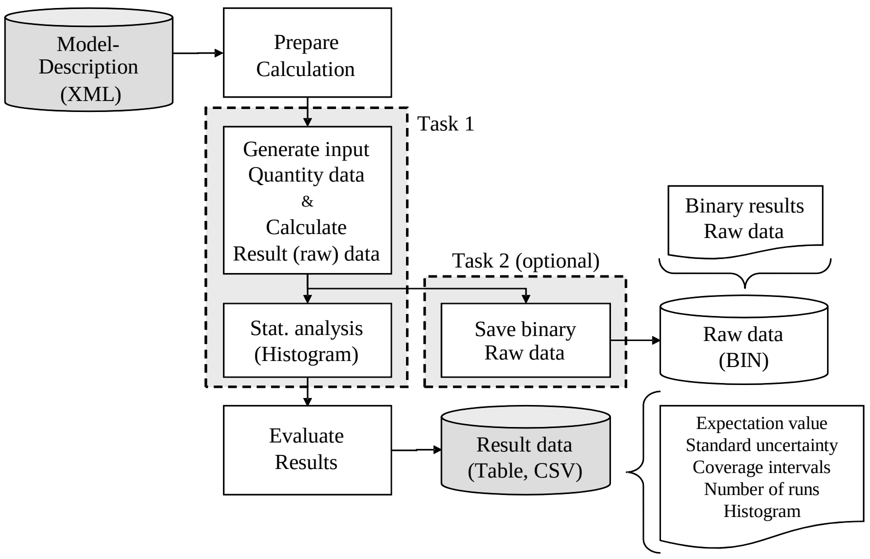

Figure 1. Block diagram of the OMCE processing task structure.

On modern personal computers, the time consuming section of the program are marked as Task 1 and Task 2 in Figure 1. These are the sections which should be improved by parallel computing. Since saving of the binary data is usually not required, we will limit our effort to implement parallel computing in Task 1.

2. Client/Server Model

Different communication concepts can be used to establish parallel computing. Esward *et. al.* [5] used an approach with distributed executables and virtually no communication between the nodes during the simulation. This approach is useful in cases when the calculation of the result is computationally the most time consuming part and the communication effort to build the histogram and merge the results can be ignored compared to the calculation of results. In this case even a distributed random generator is not really needed. The random numbers could be generated centrally and communicated with little extra effort.

In cases when the total computation effort is not several hours but a few minutes, it is still useful to improve the overall computation time. This allows increasing the number of simulation runs dramatically or enables the usage of MC simulation in the context of an interactive program. This is especially useful since multiple core systems are available already at the desktop. In this context the communication aspect becomes important.

A well established paradigm in network computing is the client server model. Several (lightweight) clients require computational tasks and delegate the tasks to a central server specialized for the task, which returns the results to the clients as soon as they are produced by the server. Typical applications are cases which involve huge amount of central data (databases).

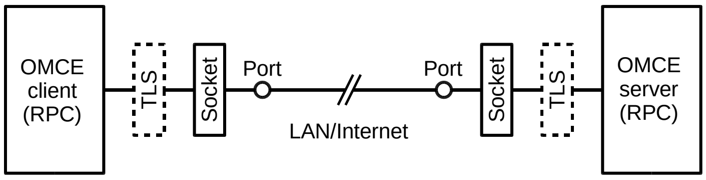

Figure 2. Block diagram of the OMCE client/server communication structure.

The introduction of the client server model to the OMCE simulator as shown in Figure 2 allows the transparent usage of the server resources by the clients. The communication is TCP/IP socket based and bound to a pre-defined port at the server side. An optional TLS layer supports secure communication and client authorization.

3. Parallel Computing

The client/server model does not automatically employ parallel computing since the client usually waits until the server completes the task (the MC simulation). But parallel computing can be realized if one MC simulation job is split up in smaller sub-tasks and several servers are employed with the sub-jobs simultaneously. Since the client requires virtually no computing resources while the server is running, handling a large number of client interfaces simultaneously is easily possible.

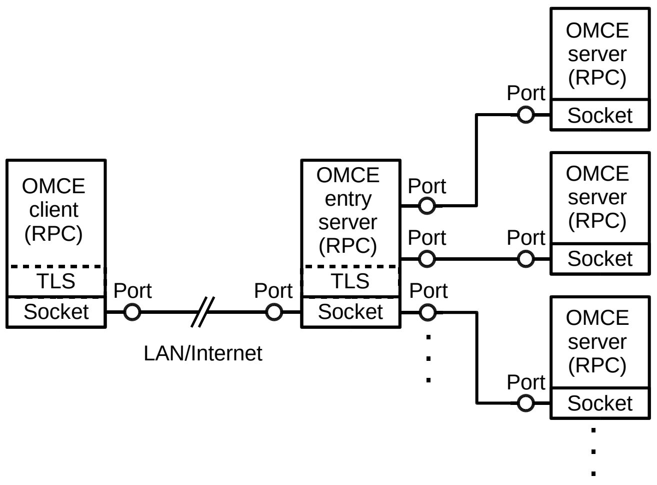

Figure 3. Block diagram of the OMCE parallel computing network using a client/server communication model.

Figure 3 shows the structure of the parallel computing network as it has been realized with the OMCE. An additional network entry server has been added which provides a standard server interface to the client, but does not perform any simulation. Instead, the entry server maintains a number of client interfaces to single task simulation servers which do the simulation. The entry server splits up the job in simulation sub-jobs and starts the servers, waits for them to finish and merges the result data. It should be noted that the same specification can be used for all interfaces between clients and servers. The entry server is reentrant and managed as a reusable server pool.

3.1. *Remote Procedure Call (RPC)*

The protocol which is used for the communication between client and server is called RPyC [6] and is an implementation of a remote procedure call (RPC) protocol in Python. The idea of RPC is to allow calling functions on a remote computer transparently as if they would be available on the local machine. The RPC protocol transparently handles all transportation aspects for parameters and the return values.

RPyC is symmetric and supports remote objects which allow using objects on the remote computer as if they would be local. The RPyC protocol is also independent from the operating system and the machine hardware allowing client and server to run on different platforms (e.g. Windows, Linux, Solaris, etc.).

A special case is the use of RPC for between task communication when client and server are running on the same computer system. Although a socket base communication needs more overhead compared to specialized concepts like the message passing interface (MPI), the RPC communication is still efficient in case of MC simulations. Therefore the concept can be used for modern desk top PCs employing quad and six core CPUs.

In principle the RPC protocol blocks further execution until the remote function is executed completely. This can significantly delay the simulation especially in case of a slow network. FIFO buffers are used to compensate this effect whenever possible allowing the simulation server to continue its task without waiting for output operations to be completed.

3.2. *Seeding*

The statistical quality of the MC simulation is highly dependent on the quality of the random number generation. The basis for all distributions used during the simulation is an equally distributed random generator which generates values within a specified interval (usually between 0 and 1). The challenge for distributed random number generation is to produce high quality sequences which do not even partly overlap. Any overlapping would create an unknown correlation and would compromise the validity of the statistical evaluation.

The OMCE uses the Mersenne Twister (MT) algorithm for generating random numbers. The MT algorithm has a large period ($2^{19937} - 1$). Because of this very large period, it is very unlikely that sequences of two generators overlap if they are seeded differently. In the context of the MT algorithm with a small number of parallel generators the problem of independent random number generation can be reduced to a seeding problem.

Esward *et. al.* [5] suggested the use of a parallel Wichmann–Hill random generator. For this generator it is guaranteed that the parallel sequences do not overlap. Matsumoto *et. al.* [7] suggested the use of a special technique to create independent MT random number generators by encoding a unique number into the characteristic polynomial of the individual generators. These techniques should be used if the number of parallel MC tasks is large (>100).

The unique seeding is done by the network entry server and becomes part of the sub-job. The present method uses a series of seeds with a fixed offset between the seeds. The start value is either derived from the clock or given by the user. This method does not guarantee identical results for identical seeds between sequential and parallel evaluation, but it guarantees identical results between identically seeded parallel simulations.

3.3. *Error Handling*

An important aspect for the practical usability of the system is the error handling. The Errors a distributed system is confronted with can be divided in two categories. All Errors which are also possible in a sequential system belong to the first category. This includes all kind of I/O-errors, data-errors or logical errors. The simulation is stopped if such errors occur during the execution on one of the servers.

Errors which are only possible in parallel systems belong to the second category. The OMCE can recover from a number of these errors during a running simulation job as long as the connection to the network entry server is available and the entry server is running. Examples of these errors are break-down of the communication to single servers or shutdown of single servers. The communication between client and server is constantly monitored and is reestablished if necessary.

4. Test and Performance Measurements

The OMCE has been tested extensively using a number of problems where theoretical solutions were known upfront to prove that the simulator works correctly.

The performance of the parallel version has been studies using the simple additive model

$$Y = \sum_{i=1}^{n} X_i \; . \qquad (1)$$

The X_i are distributed normally $N(0,1)$. The computational effort of this problem can be easily increased by increasing n or by increasing the number of MC runs. The tests were run using a client running Windows 7 which was connected (via Gigabit Ethernet) to a network entry server managing 40 simulation servers run by 4 x Opteron 6174 (2.2 GHz) CPUs (48 cores) under Ubuntu 10.04 64 Bit Linux. Linux ensures that the simulation servers are run by different CPU cores as long as the number of servers does not exceed the number of cores.

Figure 4 shows the performance of the overall system for a medium size job which runs about 60 seconds using one single task simulator. The Performance depends on the number of CPU cores used. For this medium size job the minimal overall execution time is about 7 seconds employing 27 CPU cores.

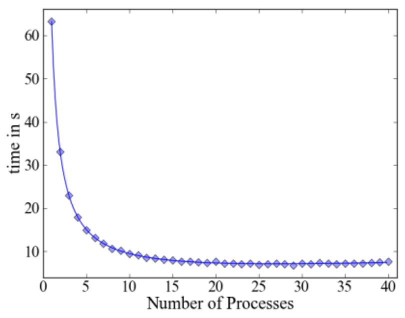

Figure 4. Performance of the overall system for a medium size job.

4.1. *Parallel Slowdown*

Figure 5 shows the performance of the same system as a function of employed CPU cores simulating a smaller job. Up to about 10 CPU cores the overall execution time is decreased, but beyond that point the execution time increases almost linearly with the number of CPUs used for the job.

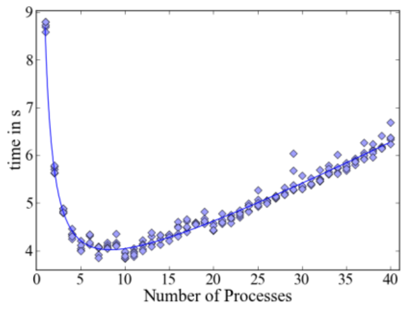

Figure 5. The performance of the overall system for a small job showing parallel slow-down.

This effect is known as the parallel slowdown. It is caused by the necessary overhead to manage multiple CPU cores. Figure 5 suggests that the overhead is almost proportional to the number of CPU cores. Studies with different problems show that the performance diagram is dependent on the problem but it usually follows a similar function as shown in Figure 4 and 5.

4.2. *Performance Model*

For a given problem the performance or overall execution time of the parallel simulation can be estimated as a function of the number of processing cores *n* used during the simulation by the following equation

$$t(n) = t_0 + \frac{t_1}{n} + t_2 \cdot n + t_3 \cdot n^2 , \tag{2}$$

where t_0, t_1, t_2 and t_3 are problem dependent timing constants. The timing constants can be found by linear regression fitting the data from multiple simulations to the model given in Equation (2). Practical studies have shown that the timing constant t_3 is very small. This can be explained by the fact that the communication structure is strictly a star with no inter-communication between the clients. The extra communication overhead is therefore proportional to the number of clients used during the simulation. Therefore Equation (2) can be simplified to

$$t^*(n) = t_0 + \frac{t_1}{n} + t_2 \cdot n . \tag{3}$$

Based on this model the minimum overall execution time can be estimated using

$$\frac{\partial t^*(n_{\max})}{\partial n_{\max}} = t_2 - \frac{t_1}{n_{\max}^2} = 0 , \tag{4}$$

which leads to

$$n_{\max} = \sqrt{\frac{t_1}{t_2}} \tag{5}$$

as the maximum number of processing cores that can be used before the parallel slow-down dominates the processing. It should be noted that this is the absolute limit for the number of processing cores, which should be used for a given problem. A significantly smaller number of cores usually results in almost the same performance.

Table 1. Expected execution time for a midsized job (Figure 4).

No. of cores	Total exec. time in s	Relative difference to minimum
12	8.68	+21.57 %
15	7.92	+10.93 %
18	7.50	+5.03 %
22	7.23	+1.18 %
27	7.14	minimum
32	7.22	+1.10 %

Table 1 shows the expected execution time (based on Equation (2)) for the midsized test job with different number of CPU cores employed. A significant reduction of the execution time from about 60 seconds down to 8.7 seconds can be achieved by employing 12 CPU cores which is less than 50% of the number needed to achieve the shortest overall execution time (7.14 s). The difference would be almost unrecognizable by the end user.

The minimum job size for a sub-job is one simulation block which has usually a size of 10000 runs. The block size can be reduced but the presented concepts are not optimized for cases where the execution time is dominated by the evaluation of the model for example for iteration problems. In these cases a massively parallel system (>100 cores) should be employed using a centralized random number generator and sophisticated job scheduling similar to the solution presented in [5].

4.3. *Estimating the Performance Parameters*

The aim of the OMCE is a general purpose simulator which can be used for all kinds of problems in metrology. To avoid parallel slow-down in the parallel version it is useful to limit the number of CPU cores used for a specific job. The maximum number of CPU cores can be evaluated from Equation (5). Unfortunately the timing parameters t_1 und t_2 depend on the simulation problem. Studies with different problems have shown that the value of the timing parameter t_2 varies only marginally between jobs and can be fixed for the test system to a value of 100 ms.

The timing parameter t_1 is basically proportional to the total execution time needed by a single task simulator. But it would be useless to run a complete simulation to estimate this parameter to make sure that the number of CPU cores is not exceeding the parallel slowdown limit.

Studies have shown that the timing parameter t_1 can be estimated from the execution time t_b needed to simulate a fraction (one block) of the total simulation. After the execution of one simulation block, the maximum number of CPU cores can be evaluated based on t_b and t_2 as

$$n_\mathrm{max} \approx p_2 \sqrt{\frac{p_1 \cdot m \cdot t_\mathrm{b}}{t_2}}\ , \tag{6}$$

with m being the total number of simulation blocks and p_1 and p_2 being scaling factors to adjust the heuristic to a given hardware system.

The network entry server starts a predefined number of simulation servers to execute one block. The execution is timed and the network entry server calculates the maximum number of servers using Equation (6). The number of servers gets adjusted and the simulation is continued.

5. Conclusions

Parallel computing based on a client server communication model can be used to speed up Monte Carlo simulation. The presented concepts are especially suitable to employ multiple core CPUs with midsized problems. Applications are those cases where the model has a large number of quantities or when an increased number of simulation runs ($>10^6$) is necessary.

The presented solution automatically adapts the number of CPU cores to the size of the problem by using a heuristic based on the execution of one simulation block to prevent parallel slowdown.

Fault tolerant communication and error recovery techniques are the basis for a robust simulation service which can be transparently integrated into a desktop environment.

Acknowledgments

We would like to thank Felix Maibaum at PTB for supporting our work by providing access to a Linux multi-core system for implementation and tests of the OMCE parallel support.

Disclaimer

Some product names are identified in this paper in order to specify the test environment adequately. Such identification are not intended to imply recommendation or endorsement by NIST or PTB, nor is it intended to imply that the products identified are necessarily the best available for the purpose.

References

1. Guide to the Expression of Uncertainty in Measurement, International Organization for Standardization, 1995
2. Guide to the Expression of Uncertainty in Measurement Supplement 1: Propagation of distributions using a Monte Carlo method, JCGM 101:2008
3. Open Monte Carlo Engine (OMCE) – Python based Open Source program, developed by R. Kessel at NIST (e-mail: ruediger.kessel@gmail.com).
4. Python – Programming Language, open source, http://www.python.org
5. T J Esward, A de Ginestous, P M Harris, I D Hill, S G R Salim, I M Smith, B A Wichmann, R Winkler and E R Woolliams: A Monte Carlo method for uncertainty evaluation implemented on a distributed computing system. Metrologia 44 (2007), pp 319–326.
6. RPyC – Remote Python Call, open source, http://rpyc.sourceforge.net

7. M Matsumoto, T Nishimura: Dynamic creation of pseudorandom number generators, Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, 2000, pp 56–69.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf