Grid-based Monte Carlo Application

Yaohang Li and Michael Mascagni

Department of Computer Science and School of Computational Science and Information Technology
Florida State University
Tallahassee, FL 32306-4530
{yaohanli, mascagni}@cs.fsu.edu

Abstract. Monte Carlo applications are widely perceived as computationally intensive but naturally parallel. Therefore, they can be effectively executed on the grid using the dynamic bag-of-work model. We improve the efficiency of the subtask-scheduling scheme by using an *N-out-of-M* strategy, and develop a Monte Carlo-specific lightweight checkpoint technique, which leads to a performance improvement for Monte Carlo grid computing. Also, we enhance the trustworthiness of Monte Carlo grid-computing applications by utilizing the statistical nature of Monte Carlo and by cryptographically validating intermediate results utilizing the random number generator already in use in the Monte Carlo application. All these techniques lead to a high-performance grid-computing infrastructure that is capable of providing trustworthy Monte Carlo computation services.

1. Introduction

Grid computing is characterized by large-scale sharing and cooperation of dynamically distributed resources, such as CPU cycles, communication bandwidth, and data, to constitute a computational environment [1]. In the grid's dynamic environment, from the application point of view, two issues are of prime import: performance – how quickly the grid-computing system can complete the submitted tasks, and trustworthiness – that the results obtained are, in fact, due to the computation requested. To meet these two requirements, many grid-computing or distributed-computing systems, such as Condor [2], HARNESS [3], Javelin [4], Globus [5], and Entropia [7], concentrate on developing high-performance and trust-computing facilities through system-level approaches. In this paper, we are going to analyze the characteristics of Monte Carlo applications, which are a potentially large computational category of grid applications, to develop approaches to address the performance and trustworthiness issues from the application level.

The remainder of this paper is organized as follows. In Section 2, we analyze the characteristics of Monte Carlo applications and develop a generic grid-computing paradigm for Monte Carlo computations. We discuss how to take advantage of the characteristics of Monte Carlo applications to improve the performance and trustworthiness of Monte Carlo grid computing in Section 3 and Section 4,

respectively. Finally, Section 5 summarizes our conclusions and future research directions.

2. Grid-based Monte Carlo Applications

Among grid applications, those using Monte Carlo methods, which are widely used in scientific computing and simulation, have been considered too simplistic for consideration due to their natural parallelism. However, below we will show that many aspects of Monte Carlo applications can be exploited to provide much higher levels of performance and trustworthiness for computations on the grid. According to word of mouth, about 50% of the CPU time used on supercomputers at the U.S. Department of Energy National Labs is spent on Monte Carlo computations. Unlike data-intensive applications, Monte Carlo applications are usually computation intensive [6] and they tend to work on relatively small data sets while often consuming a large number of CPU cycles. Parallelism is a way to accelerate the convergence of a Monte Carlo computation. If N processors execute N independent copies of a Monte Carlo computation, the accumulated result will have a variance N time smaller than that of a single copy. In a distributed Monte Carlo application, once a distributed task starts, it can usually be executed independently with almost no inter-process communication. Therefore, Monte Carlo applications are perceived as naturally parallel, and they can usually be programmed via the so-called dynamic *bag-of-work* model. Here a large task is split into smaller independent subtasks and each are then executed separately. Effectively using the dynamic *bag-of-work* model for Monte Carlo requires that the underlying random number streams in each subtask be independent in a statistical sense. The SPRNG (Scalable Parallel Random Number Generators) library [11] was designed to use parameterized pseudorandom number generators to provide independent random number streams to parallel processes. Some generators in SPRNG can generate up to 2^{31}-1 independent random number streams with sufficiently long period and good quality [13]. These generators meet the random number requirements of most Monte Carlo grid applications.

The intrinsically parallel aspect of Monte Carlo applications makes them an ideal fit for the grid-computing paradigm. In general, grid-based Monte Carlo applications can utilize the grid's *schedule service* to dispatch the independent subtasks to the different nodes [15]. The execution of a subtask takes advantage of the *storage service* of the grid to store intermediate results and to store each subtask's final (partial) result. When the subtasks are done, the *collection service* can be used to gather the results and generate the final result of the entire computation. Fig. 1 shows this generic paradigm for Monte Carlo grid applications.

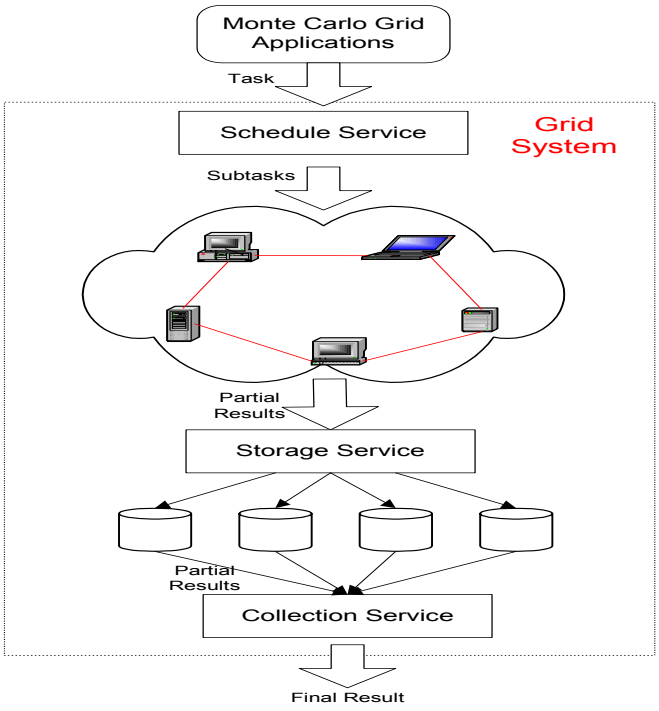

Fig. 1. Monte Carlo Application in a Grid System

The inherent characteristics of Monte Carlo applications motivate the use of grid computing to effectively perform large-scale Monte Carlo computations. Furthermore, within this Monte Carlo grid-computing paradigm, we can use the statistical nature of Monte Carlo computations and the cryptographic aspects of random numbers to reduce the wallclock time and to enforce the trustworthiness of the computation.

3. Improving the Performance of Grid-based Monte Carlo Computing

3.1 *N-out-of-M* Strategy

The nodes that provide CPU cycles in a grid system will most likely have computational capabilities that vary greatly. A node might be a high-end supercomputer, or a low-end personal computer, even just an intelligent widget. In addition, these nodes are geographically widely distributed and not centrally manageable. A node may go down or become inaccessible without notice while it is working on its task. Therefore, a slow node might become the bottleneck of the whole

computation if the assembly of the final result must wait for the partial result generated on this slow node. A delayed subtask might delay the accomplishment of the whole task while a halted subtask might prevent the whole task from ever finishing. To address this problem, system-level methods are used in many grid or distributed-computing systems. For example, Entropia [7] tracks the execution of each subtask to make sure none of the subtasks are halted or delayed. However, the statistical nature of Monte Carlo applications provides a shortcut to solve this problem at the application level.

Suppose we are going to execute a Monte Carlo computation on a grid system. We split it into N subtasks, with each subtask based on its unique independent random number stream. We then schedule each subtask onto the nodes in the grid system. In this case, the assembly of the final result requires all the N partial results generated from the N subtasks. Each subtask is a "key" subtask, since the suspension or delay of any one of these subtasks will have a direct effect on the completion time of the whole task.

When we are running Monte Carlo applications, what we really care about is how many random samples (random trajectories) we must obtain to achieve a certain, predetermined, accuracy. We do not much care which random sample set is estimated, provided that all the random samples are independent in a statistical sense. The statistical nature of Monte Carlo applications allows us to enlarge the actual size of the computation by increasing the number of subtasks from N to M, where $M > N$. Each of these M subtasks uses its unique independent random number set, and we submit M instead of N subtasks to the grid system. Therefore, M bags of computation will be carried out and M partial results may be eventually generated. However, it is not necessary to wait for all M subtasks to finish. When N partial results are ready, we consider the whole task for the grid system to be completed. The application then collects the N partial results and produces the final result. At this point, the grid-computing system may broadcast abort signals to the nodes that are still computing the remaining subtasks. We call this scheduling strategy *the N-out-of-M strategy*. In the *N-out-of-M strategy* more subtasks than are needed are actually scheduled, therefore, none of these subtasks will become a "key" subtask and we can tolerate at most $M - N$ delayed or halted subtasks.

Fig. 2 shows an example of a distributed Monte Carlo computation using the "*6-out-of-10*" strategy. In this example, 6 partial results are needed and 10 subtasks are actually scheduled. During the computation, one subtask is suspended for some unknown reason. In addition, some subtasks have very short completion time while others execute very slowly. However, when 6 of the subtasks are complete, the whole computation is complete. The suspended subtask and the slow subtasks do not affect the completion of the whole computational task.

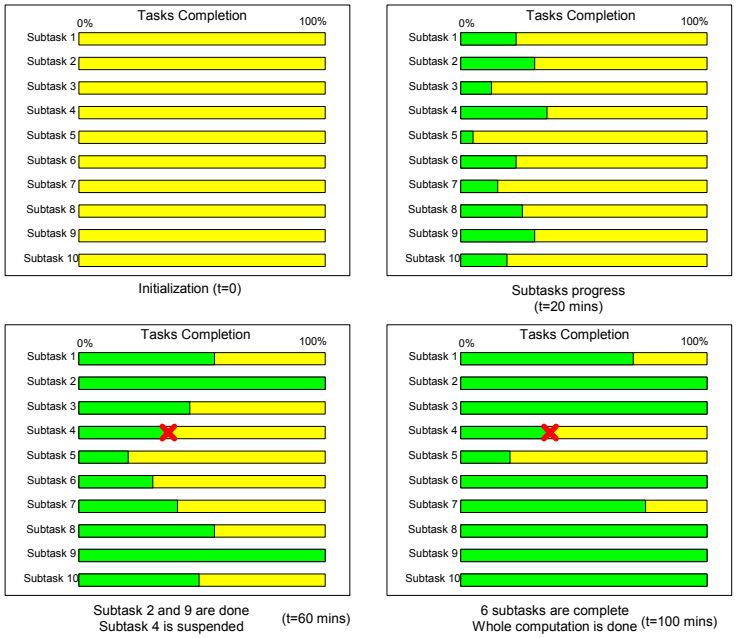

Fig. 2. Example of the "*6-out-of-10*" Strategy with 1 Suspended and 3 "Slow" Subtasks

In Monte Carlo applications, N is determined by the application and it depends on the number of random samples or random trajectories needed to obtain a predetermined accuracy. The problem is thus how to choose the value M properly. A good choice of M can prevent a few subtasks from delaying or even halting the whole computation. However, if M is chosen too large, there may be little benefit to the computation at the cost of significantly increasing the workload of the grid system. The proper choice of M in the *N-out-of-M strategy* can be determined by considering the average job-completion rate in the grid system. Suppose p is the completion probability of subtasks up to time t in the grid system. Clearly, $M*p$ should be approximately N, i.e., the fraction of the M subtasks finished should equal to N. Thus a good choice is $M = \lceil N/p \rceil$. Note, if we know something about $p(t)$, the time-dependent completion probability, we can use this same reasoning to also help specify the approximate running time.

We model the *N-out-of-M* strategy based on a binomial model. Assume that the probability of a subtask completing by time t is given $p(t)$. Also assume that $p(t)$ describes the aggregate probability over the pool of nodes in the grid, i.e., it could be measured by computing the empirical frequencies of the completion times over the pool. Then the probability that exactly N out of M subtasks are complete at time t is given by

$$P_{Exactly-N-out-of-M}(t) = \binom{M}{N} p^N(t) \times (1 - p(t))^{M-N}, \quad \textbf{(1)}$$

and so the probability that at least N subtasks are complete is given by

$$P_{N-out-of-M}(t) = \sum_{i=N}^{M} \binom{M}{i} p^i(t) \times (1 - p(t))^{M-i} . \quad (2)$$

The old strategy can be thought of as "*N-out-of-N*" which has probability given by

$$P_{N-out-of-N}(t) = p^N(t) . \quad (3)$$

Fig. 3 shows an approximate sketch of $P(t)_{N\text{-}out\text{-}of\text{-}M}$, $p(t)$, and $P(t)_{N\text{-}out\text{-}of\text{-}N}$ ($p(t)$ can be either below $P(t)_{N\text{-}out\text{-}of\text{-}M}$ or above $P(t)_{N\text{-}out\text{-}of\text{-}M}$, depending on the value of N and M). As time goes on, the *N-out-of-M* strategy always has a higher probability of completion than the *N-out-of-N* strategy, although they all converge to 1.0 at large times.

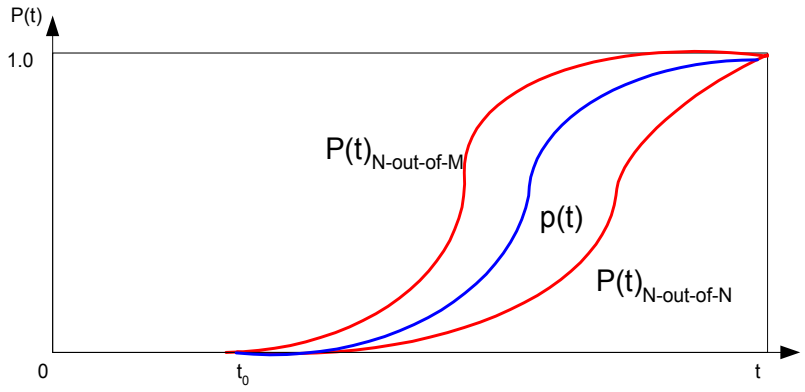

Fig. 3. Sketch of $P_{N\text{-}out\text{-}of\text{-}M}$, $p(t)$, and $P_{N\text{-}out\text{-}of\text{-}N}$

Also notice that the Monte Carlo computation using the *N-out-of-M strategy* is reproducible, because we know exactly which N out of M subtasks are actually involved and which random numbers were used. Thus each of these N subtasks can be reproduced later. However, if we want to reproduce all of these N subtasks at a later time on the grid system, the *N-out-of-N* strategy must be used!

One drawback of the *N-out-of-M* strategy is we must execute more subtasks than actually needed and will therefore increase the computational workload on the grid system. However, our experience with distributed computing systems such as Condor and Javelin shows that most of the time there are more nodes providing computing services available in the grid system than subtasks. Therefore, properly increasing the computational workload to achieve a shorter completion time for a computational task should be an acceptable tradeoff in a grid system.

3.2 Lightweight Checkpointing

A subtask running on a node in a grid system may take a very long time to finish. The *N-out-of-M* strategy is an attempt to mitigate the effect of this on the overall running

time. However, if one incorporates checkpointing, he can directly attack reducing the completion time of the subtasks. Some grid computing systems implement a process-level checkpoint. Condor, for example, takes a snapshot of the process's current state, including stack and data segments, shared library code, process address space, all CPU states, states of all open files, all signal handlers, and pending signals [12]. On recovery, the process reads the checkpoint file and then restores its state. Since the process state contains a large amount of data, processing such a checkpoint is quite costly. Also, process-level checkpointing is very platform-dependent, which limits the possibility of migrating the process-level checkpoint to another node in a heterogeneous grid-computing environment.

Fortunately, Monte Carlo computation has a structure highly amenable to application-based checkpointing. Typically, a Monte Carlo application starts in an initial configuration, evaluates a random sample or a random trajectory, estimates a result, accumulates mean and variances with previous results, and repeats this process until some termination condition is met. Thus, to recover an interrupted computation, a Monte Carlo application needs to save only a relatively small amount of information. The necessary information to reconstruct a Monte Carlo computation image at checkpoint time will be the current results based on the estimates obtained so far, the current status and parameters of the random number generators, and other relevant program information like the current iteration number. This allows one to make a smart and quick application checkpoint in most Monte Carlo applications. Using XML [8] to record the checkpointing information, we can make this checkpoint platform-independent. More importantly, compared to a process checkpoint, the application-level checkpoint is much smaller in size and much quicker to generate. Therefore, it should be relatively easy to migrate a Monte Carlo computation from one node to another in a grid system. However, the implementation of application level checkpointing will somewhat increase the complexity of developing new Monte Carlo grid applications.

4. Enhancing the Trustworthiness of Grid-based Monte Carlo Computing

4.1 Distributed Monte Carlo Partial Result Validation

The correctness and accuracy of grid-based computations are vitally important to an application. In a grid-computing environment, the service providers of the grid are often geographically separated with no central management. Faults may hurt the integrity of a computation. These might include faults arising from the network, system software or node hardware. A node providing CPU cycles might not be trustworthy. A user might provide a system to the grid without the intent of faithfully executing the applications obtained. Experience with SETI@home has shown that users often fake computations and return wrong or inaccurate results. The resources in a grid system are so widely distributed that it appears difficult for a grid-computing

system to completely prevent all "bad" nodes from participating in a grid computation. Unfortunately, Monte Carlo applications are very sensitive to each partial result generated from each subtask. An erroneous partial result will most likely lead to the corruption of the whole grid computation and thus render it useless.

The following Monte Carlo integration example illustrates how an erroneous computational partial result effects the whole computation. Let us consider the following hypothetical Monte Carlo computation. Suppose we wish to evaluate integral

$$\int_0^1 \cdots \int_0^1 \frac{4x_1 x_3^2 e^{2x_1 x_3}}{(1+x_2+x_4)^2} e^{x_5+\ldots+x_{20}} x_{21} x_{22} \ldots x_{25} dx_1 \ldots dx_{25} \,. \tag{4}$$

The exact solution to 8-digits of this integral is 103.81372. In the experiment, we plan to use crude Monte Carlo on a grid system with 1,000 nodes. Table 1 tabulates the partial results from volunteer computers.

Table 1. Hypothetical Partial Results of Monte Carlo Integration Example

Subtask #	Partial Results
1	103.8999347
2	104.0002782
3	103.7795764
4	103.6894540
…	
561	89782.048998
…	
997	103.9235347
998	103.8727823
999	103.8557640
1000	103.7891408

Due to an error, the partial result returned from the node running subtask #561 is clearly bad. The fault may have been due to an error in the computation, a network communication error, or malicious activity, but that is not important. The effect is that the whole computational result ends 193.280805, considerably off the exact answer. From this example, we see that, in grid computing, the final computational result may be sensitive to each of the partial results obtained from nodes in the grid system. An error in a computation may seriously hurt the whole computation.

To enforce the correctness of the computation, many distributed computing or grid systems adapt fault-tolerant methods, like duplicate checking [10] and majority vote [16]. In these approaches, subtasks are duplicated and carried out on different nodes. Erroneous partial results can be found by comparing the partial results of the same subtask executed on different nodes. Duplicated checking requires doubling computations to discover an erroneous partial result. Majority vote requires at least three times more computation to identify an erroneous partial result. Using duplicate checking or majority vote will significantly increase the workload of a grid system.

In the dynamic *bag-of-work* model as applied to Monte Carlo applications, each subtask works on the same description of the problem, but estimates based on different random samples. Since the mean in a Monte Carlo computation is

accumulated from many samples, its distribution will be approximately normal, according to the Central Limit Theorem. Suppose $f_1, \ldots, f_i, \ldots, f_n$ are the n partial results generated from individual nodes on a grid system. The mean of these partial results is

$$\hat{f} = \frac{1}{n}\sum_{i=1}^{n} f_i , \tag{5}$$

and we can estimate its standard error, s, via the following formula

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(f_i - \hat{f})^2} . \tag{6}$$

Specifically, the Central Limit Theorem states that $\hat{f}$ should be distributed approximately as a student-t random variable with mean $\hat{f}$, standard deviation $s/\sqrt{n}$, and n degrees-of-freedom. However, since we usually have n, the number of subtasks, chosen to be large, we may instead approximate the student-t distribution with the normal. Standard normal confidence interval theory states that with 68% confidence that the exact mean is within 1 standard deviation of $\hat{f}$, with 95% confidence within 2 standard deviations, and 99% confidence within 3 standard deviations. This statistical property of Monte Carlo computation can be used to develop an approach for validating the partial results of a large grid-based Monte Carlo computation.

Here is the proposed method for distributed Monte Carlo partial result validation. Suppose we are running n Monte Carlo subtasks on the grid, the ith subtask will eventually return a partial result, f_i. We anticipate that f_i are approximately normally distributed with mean, $\hat{f}$, and standard deviation, $\sigma = s/\sqrt{n}$. We expect that about one of the f_i in this group of n to lie outside a normal confidence interval with confidence $1 - 1/n$. In order to choose a confidence level that permits events we expect to see, statistically, yet flag events as outliers requires us to choose a multiplier, c, so that we flag events that should only occur in a group of size cn. The choice of c is rather subjective, but $c = 10$ implies that in only 1 in 10 runs of size n we should expect to find an outlier with confidence $1 - 1/10n$. With a given choice of c, one computes the symmetric normal confidence interval based on a confidence of $\alpha\% = 1 - 1/cn$. Thus the confidence interval is $[\hat{f} - Z_{\alpha/2}\sigma,\ \hat{f} + Z_{\alpha/2}\sigma]$, where $Z_{\alpha/2}$ is unit normal value such that $\int_0^{Z_{\alpha/2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = \frac{\alpha}{2}$. If f_i is in this confidence interval, we can consider this partial result as trustworthy. However, if f_i falls out of the interval, which may happen merely by chance with a very small probability, this particular partial result is suspect. We may either rerun the subtask that generated the suspicious partial result on another node for further validation or just discard it (if using the *N-out-of-M* strategy).

Let us now come back to the previous Monte Carlo integration example. We performed an experiment by running 1,000 subtasks for evaluating the integral described in the Monte Carlo integration example on a Condor pool [14]. Fig. 4 shows the distribution of all the generated partial results: 677 partial results are

located within 1 standard deviation of the mean, 961 partial results within 2 standard deviations, and 999 of the 1,000 partial results within 3 standard deviations. If a hypothetical partial result happens as the one (#561) in the Monte Carlo integration example, the outlier lies 30 standard deviations to the right of the mean. As we know from calculating the confidence interval, we have α = 99.9999999999% within 7 standard deviations. A outlier falling outside of 7 standard deviations of the mean will be expected to happen by chance only once in 10^9 experiments. Therefore, the erroneous partial result of #561 in the Monte Carlo integration example will easily be captured and flagged as abhorrent.

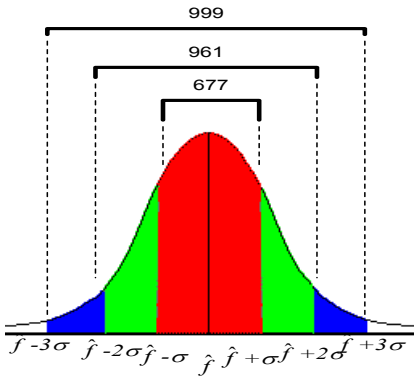

Fig. 4. Partial Result Distribution in Monte Carlo Integration Example

This Monte Carlo partial result validation method supplies us with a way to identify suspicious results without running more subtasks. This method assumes that the majority of the nodes in grid system are "good" service providers, which can correctly and faithfully execute their assigned task and transfer the result. If most of the nodes are malicious, this validation method may not be effective. However, experience has shown that the fraction of "bad" nodes in volunteered computation is very small.

4.2 Intermediate value checking

Usually, a grid-computing system compensates the service providers to encourage computer owners to supply resources. Many Internet-wide grid-computing projects, such as SETI@home [9], have the experience that some service providers didn't faithfully execute their assigned subtasks. Instead they attempt to provide bogus partial result at a much lower personal computational cost in order to obtain more benefits. Checking whether the assigned subtask from a service provider is faithfully carried out and accurately executed is a critical issue that must be addressed by a grid-computing system.

One approach to check the validity of a subtask computation is to validate intermediate values within the computation. Intermediate values are some quantities generated within the execution of the subtask. To the node that runs the subtask, these values will be unknown until the subtask is actually executed and reaches a specific

point within the program. On the other hand, to the clever application owner, certain intermediate values are either pre-known or are very easy to generate. Therefore, by comparing the intermediate values and the pre-known values, we can control whether the subtask is actually faithfully carried out or not. Monte Carlo applications consume pseudorandom numbers, which are generated deterministically from a pseudorandom number generator. If this pseudorandom number generator has a cheap algorithm for computing arbitrarily within the period, the random numbers are perfect candidates to be these cleverly chosen intermediate values. Thus, we have a very simple strategy to validate a result from subtasks by tracing certain predetermined random numbers in Monte Carlo applications.

For example, in a grid Monte Carlo application, we might force each subtask to save the value of the current pseudorandom number after every N (e.g., $N = 100{,}000$) pseudorandom numbers are generated. Therefore, we can keep a record of the Nth, $2N$th, …, kNth random numbers used in the subtask. To validate the actual execution of a subtask on the server side, we can just re-compute the Nth, $2N$th, …, kNth random numbers applying the specific generator with the same seed and parameters as used in this subtask. We then simply match them. A mismatch indicates problems during the execution of the task. Also, we can use intermediate values of the computation along with random numbers to create a cryptographic digest of the computation in order to make it even harder to fake a computational result. Given our list of random numbers, or a deterministic way to produce such a list, when those random numbers are computed, we can save some piece of program data current at that time in an array. At the same time we can use that random number to encrypt the saved data and incorporate these encrypted values in a cryptographic digest of the entire computation. At the end of the computation the digest and the saved values are then both returned to the server. The server, through cryptographic exchange, can recover the list of encrypted program data and quickly compute the random numbers used to encrypt them. Thus, the server can decrypted the list and compare it to the "plaintext" versions of the same transmitted from the application. Any discrepancies would flag either an erroneous or faked result. While this technique is certainly not a perfect way to ensure correctness and trustworthiness, a user determined on faking results would have to scrupulously analyze the code to determine the technique being used, and would have to know enough about the mathematics of the random number generator to leap ahead as required. In our estimation, surmounting these difficulties would far surpass the amount of work saved by gaining the ability to pass off faked results as genuine.

5. Conclusions

Monte Carlo applications generically exhibit naturally parallel and computationally intensive characteristics. Moreover, we can easily fit the dynamic *bag-of-work* model, which works so well for Monte Carlo applications, onto a grid system to implement grid-based Monte Carlo computing. Furthermore, we may take advantage of the statistical nature of Monte Carlo calculations and the cryptographic nature of random

numbers to enhance the performance and trustworthiness of this Monte Carlo grid-computing infrastructure at the application level.

The next phase of our research will be to develop a Monte Carlo grid toolkit, using the techniques described in this paper, to facilitate the development of grid-based Monte Carlo applications. At the same time, we will also try to execute more real-life Monte Carlo applications on our developing grid system.

References

1. I. Foster, C. Kesselman, and S. Tueske, "The Anatomy of the Grid," International Journal of Supercomputer Applications, **15**(3), 2001.
2. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.
3. Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam, "HARESS: a next generation distributed virtual machine," Journal of Future Generation Computer Systems, (15), Elsevier Science B. V., 1999.
4. B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, "Javelin: Internet-Based Parallel Computing Using Java," Concurrency: Practice and Experience, **9**(11): 1139 - 1160, 1997.
5. I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," International Journal of Supercomputer Applications, **11**(2), 1997.
6. A. Srinivasan, D. M. Ceperley, and M. Mascagni, "Random Number Generators for Parallel Applications," to appear in Monte Carlo Methods in Chemical Physics, D. Ferguson, J. I. Siepmann and D. G. Truhlar, editors, Advances in Chemical Physics series, Wiley, New York, 1997.
7. Entropia website, http://www.entropia.com/entropia_platform.asp.
8. XML website, http://www.xml.org.
9. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home-Massively distributed computing for SETI," Computing in Science and Engineering, **v3n1**, 81, 2001.
10. C. Aktouf, O.Benkahla, C.Robach, and A. Guran, "Basic Concepts & Advances in Fault-Tolerant Computing Design," World Scientific Publishing Company, 1998.
11. M. Mascagni, D. Ceperley, and A. Srinivasan, "SPRNG: A Scalable Library for Pseudorandom Number Generation," ACM Transactions on Mathematical Software, in the press, 2000.
12. M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for High Throughput Computing," SPEEDUP Journal, **11**(1), 1997.
13. SPRNG website, http://sprng.cs.fsu.edu.
14. Condor website, http://www.cs.wisc.edu/condor.
15. R. Buyya, S. Chapin, and D. DiNucci, "Architectural Models for Resource Management in the Grid," the First IEEE/ACM International Workshop on Grid Computing (GRID 2000), Springer Verlag LNCS Series, Germany, Bangalore, India, 2000.
16. L. F. G. Sarmenta, "Sabotage-Tolerance Mechanisms for Volunteer Computing Systems," ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), Brisbane, Australia, May, 2001.