Parallel Quasi-Monte Carlo Methods on a Heterogeneous Cluster

Giray Ökten[1] and Ashok Srinivasan[2]

[1] Claremont Research Institute of Applied Mathematical Sciences, Claremont Graduate University, USA
[2] University of California at Santa Barbara, USA

Abstract. We introduce a new parallelization scheme for low-discrepancy sequences. Our approach is similar to the parameterization approach used for pseudorandom number generators. We present a theoretical analysis of this scheme and compare it numerically with the conventional blocking and leap-frogging parallelization strategies, when they are applied to problems from option pricing and transport theory. The numerical results suggest that our scheme might be very useful especially in distributed computing environments with unreliable and heterogeneous clusters.

1 Parallel Monte Carlo

Monte Carlo computations are popular on parallel computers, since these problems are generally "embarrassingly" parallel. Identical algorithms are run almost independently on each processor, except that the random number sequences used are different on each processor. The results from all the processors are combined to get the final answer. Since the results from each processor differ only due to the different random number sequences used, it is important to ensure that the sequences on different processors are uncorrelated.

We consider the following extreme situation to illustrate possible pitfalls in parallel Monte Carlo computations. Let us assume that we use N random numbers on each of the m processors in a parallel Monte Carlo computation, but use the same random sequence. Then the results will be identical on all the processors, wasting all the parallelism of the computing environment. Although such an extreme case will not arise in general, it nevertheless illustrates how poor parallelization strategies for pseudorandom number generators can result in an essentially very inefficient use of the parallel computing environment. In general, poor parallelization strategies for pseudorandom number generators can result in the answers being incorrect.

The parallelization strategies for pseudorandom number generators used today can be broadly classified as (i) Cycle Division, where different portions of a single random number cycle are divided among different processors (examples of this technique are leap-frog and blocking), and (ii) parameterization, where inherently independent sequences are placed on different processors. Further details on parallel Monte Carlo issues can be found in [23,13,6,7].

One of the disadvantages of the cycle division methods is that long range correlations between random numbers in the original sequence can turn into short range correlations within the sequence on each processor (in the leap frog scheme), or between the sequences on different processors (in the blocking scheme). Furthermore, there have been several studies, for example [8,9], that have detected long range correlations in popular pseudorandom number generators, and thus one has to be careful with this strategy.

The SPRNG library [22,13] uses parameterization techniques in its pseudorandom number generators, and we use SPRNG in those numerical experiments of the paper that require comparisons with a parallel pseudorandom number generator.

2 Parallel Quasi-Monte Carlo

While there has been much work done on the issue of parallel pseudorandom number generators, there have been relatively fewer studies on the issues that arise in parallelizing low-discrepancy sequences. When computations are carried out on a heterogeneous cluster of workstations, there are certain additional issues that arise. For example, different processors may run at different speeds, and some processors may even fail in an unreliable system. We would like to be able to combine the results of the computations of all the processors that have not failed, and get an estimate that is roughly as accurate as that of a sequential computation with the same number of simulations.

Two schemes have been proposed in recent times [3,16,17,12] for parallelizing low-discrepancy sequences, which use a strategy analogous to that of cycle division with parallel pseudorandom number generators. They assign different subsequences of a single low discrepancy sequence to different processors. We summarize the schemes below.

Let $x^0, x^1, x^2, ...$ be an s-dimensional low-discrepancy sequence where each element x^i is an s-dimensional vector. (i) *Leap frog:* If we have m processors, then in the leap frog scheme we assign the sequence $x^i, x^{i+m}, x^{i+2m}, ...$ to processor i, $0 \leq i < m$. (ii) *Blocking:* In this scheme, we assign blocks of contiguous elements from a single sequence to each processor. Therefore processor i gets the sequence $x^{iB}, x^{iB+1}, x^{iB+2}, ...$ where B is the size of each block.

Both of the schemes mentioned above require the ability to "jump ahead" in a sequence, in order to have an efficient implementation. Such facilities were implemented for the Sobol' sequence by Bromley [3] and for the (t, s) sequences in general by Schmid and Uhl [16]. In related work, Li and Mullen [12] introduced a leap-frog scheme for (t, m, s) nets and used it to solve financial derivative problems. De Doncker et al. [10] used a distributed scheme in the master-worker paradigm for producing a quasi-random estimate.

We now discuss some properties of these schemes. (i) If each processor runs at the same speed, then we may expect them to consume the same number of elements of each sequence. If each processor uses N/m elements of the sequence, then the leap frog scheme will produce the same result as a sequential computation with N elements. With the blocking scheme, this will not happen, unless the block size happens to be identical to the number of elements consumed per processor, that is, $B = N/m$. Even in this case, the result will be identical only at the end of the computation, and not at intermediate values. In general, one does not know in advance the number of elements of the sequence that will be consumed by each processor. Hence, in order to prevent any overlapping of sequences, one chooses B to be sufficiently large so that there is little possibility of $N/m > B$. Thus results of parallel quasi-Monte Carlo with blocking scheme will typically not produce a result identical to that of a sequential run. (ii) If processors run at different speeds, then even the leap frog scheme will not give a result identical to that of a sequential run.[1] (iii) If processors run at different speeds and we use practical synchronization strategies, then the results obtained by either of the parallelization strategies will be identical to a sequential run that uses the original sequence with some "gaps". However, for a general low-discrepancy sequence, uniformity (i.e., uniform distribution modulo 1) of a sequence does not necessarily guarantee the uniformity of a subsequence, and thus there is a danger of erroneous results. Of course, if a processor fails, then the same situation applies. This can be considered as the limiting case of a processor with zero speed. Additional discussion of the leap-frog scheme and the disadvantages of Bromley's leap-frog approach can be found in [17].

3 A Parameterized Parallel Quasi-Monte Carlo Approach

We describe in this section a new parallel quasi-Monte Carlo scheme based on placing elements of a class of low-discrepancy sequences on each processor of the computing environment. We start with a brief discussion of Monte Carlo and quasi-Monte Carlo methods.

3.1 Introduction

In general, Monte Carlo algorithms are used to estimate the expected value $E[X]$ of a random variable. Independent values $X_1, ..., X_N$ of the random vari-

[1] In order to get results identical to that of a sequential run with the leap frog scheme, we would need to synchronize processors repeatedly after the completion of every, say, S iterations. However, this is wasteful, since it makes the fastest processors too effectively run at the speed of the slowest processor, and therefore this situation is not realistic. Instead, one would rather synchronize every, say, T units of time, in which case the result would not be identical to that of a sequential run.

able X are used to form the estimate $\frac{1}{N}\sum_{i=1}^{N} X_i$, which converges to $E[X]$ almost surely. In order to form these estimates, a sampling method such as inverse transform or acceptance-rejection is used, which amounts to expressing X as a function of random numbers ξ_i in $I=[0,1]$, i.e., $X=f(\xi_1,...,\xi_s)$. Therefore, to sample one value from X, we need s random numbers. Following Sobol', we will call s the *constructive dimension* of the Monte Carlo algorithm.

We say a problem is finite dimensional with dimension $s<\infty$, if the constructive dimension of the Monte Carlo algorithm used to estimate $E[X]$ is s. Similarly, a problem will be called infinite dimensional, if its constructive dimension is infinity. We emphasize that the property of having a finite dimension is determined by the Monte Carlo algorithm of the problem. There may be two Monte Carlo algorithms, attempting to estimate the expected value of the same random variable, with one having a finite constructive dimension, and the other an infinite one.

Infinite dimensional problems are often encountered in statistical physics and transport theory. In a typical infinite dimensional problem, one generates random walks from birth until termination, and there is no a priori bound on the number of steps (which determines the constructive dimension) a random walk takes until termination.

Clearly a finite dimensional problem with dimension s can be viewed as an s-dimensional numerical integration, since

$$\frac{1}{N}\sum_{i=1}^{N} X_i = \frac{1}{N}\sum_{i=1}^{N} f(\xi^{(i)}) \to \int_{I^s} f(x)dx = E[X]\ a.s., \tag{1}$$

where $\xi^{(i)}$ are random vectors uniformly distributed in I^s. An infinite dimensional problem, on the other hand, can be thought of as integration over $I^\infty=\prod_{i=1}^{\infty} I$. Equipping I^∞ with the usual product measure (Lebesgue measures on all axes), we obtain

$$\frac{1}{N}\sum_{i=1}^{N} X_i = \frac{1}{N}\sum_{i=1}^{N} f(\eta^{(i)}) \to \int_{I^\infty} f(x)dx = E[X]\ a.s., \tag{2}$$

where $\eta^{(i)}=(\eta_1^{(i)},\eta_2^{(i)},...)$.

In a quasi-Monte Carlo approach, the finite dimensional integration problem (1) can be estimated using an s-dimensional uniformly distributed sequence $q^{(1)},q^{(2)},...$ where $q^{(i)}=(q_1^{(i)},...,q_s^{(i)})$. We have

$$\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N} f(q^{(i)}) = \int_{I^s} f(x)dx, \tag{3}$$

where the convergence is deterministic. For the infinite dimensional integration problem (2), let $f_n(x_1,...,x_n)$ be a sequence of functions that are

Riemann integrable in I^n, satisfying

$$\lim_{n\to\infty} \int_{I^n} f_n(x)dx = \int_{I^\infty} f(x)dx, \tag{4}$$

and let $q^{(i)}$ be an infinite dimensional uniformly distributed sequence, i.e., a sequence whose finite dimensional projections are uniformly distributed. If we let $f_n(q^{(i)}) = f_n(q_1^{(i)}, ..., q_n^{(i)})$, then

$$\lim_{n\to\infty}\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} f_n(q^{(i)}) = \int_{I^\infty} f(x)dx, \tag{5}$$

gives a counterpart to (3) in infinite dimensions. See Sobol' ([18] and [19]) for an analysis of infinite dimensional integration.

We want to design a parallel quasi-Monte Carlo scheme which can be applied to finite as well as infinite dimensional problems. In order to simulate an infinite dimensional problem using quasi-Monte Carlo methods, one needs an infinite-dimensional uniformly distributed sequence. Among a few existing choices (Halton, Sobol'[2], Niederreiter[3], square-root), we decide to use Halton sequences and their variants in our scheme. These sequences (especially a particular scrambled Halton sequence, which will be defined later) satisfy "good" discrepancy bounds, are easy to generate, and provide surprisingly good numerical results in comparison to other low-discrepancy sequences in a variety of problems. However, the most important feature of these sequences from our perspective is that they form a class of sequences which we can distribute to different processors of the computing environment.

3.2 Parallel Halton Implementation

We will use the standard notation

$$q^{(n)} = (\phi_{b_1}(n), ..., \phi_{b_s}(n)), n \geq 0, \tag{6}$$

for the Halton sequence in the bases $b_1, ..., b_s$. Here ϕ_{b_i} is the radical-inverse function in base b_i, defined by

$$\phi_{b_i}(n) = \sum_{j=0}^{\infty} a_j(n) b_i^{-j-1}, \tag{7}$$

[2] The existing implementations of Sobol' sequence make use of additional uniformity properties by carefully selecting the direction numbers and hence they are all finite dimensional.

[3] The existing implementation of the Niederreiter sequence by Bratley, Fox and Niederreiter is also finite dimensional. However, it should be possible to write an infinite dimensional version for Niederreiter sequences that have a fixed base. If one wants to choose the base in an optimal way, however, the resulting sequence will be finite dimensional, since the knowledge of the dimension is necessary to find the optimal base that will minimize the constants in the discrepancy bounds.

where $n = \sum_{j=0}^{\infty} a_j(n) b_i^j$. The infinite-dimensional Halton sequence for a given bases sequence $\{b_1, b_2, ...\}$ is $q^{(n)} = (\phi_{b_1}(n), \phi_{b_2}(n), ...)$, $n \geq 0$. Our parallelization scheme assigns an infinite dimensional Halton sequence (or, one of its variants, as described in Sect. 3.3) to each processor. In the following, let p_i denote the ith prime number. If the computing environment has m processors, then the ith processor is assigned the Halton sequence with bases sequence $\{p_i, p_{i+m}, p_{i+2m}, ...\}$, i.e., the sequence

$$\phi_{p_i}(n), \phi_{p_{i+m}}(n), \phi_{p_{i+2m}}(n), ..., \; n = 1, 2, ..., \tag{8}$$

where $i = 1, ..., m$.

If the problem has dimension $s < \infty$, then the above infinite sequences become s-dimensional vectors for each n. Then, each processor produces an s-dimensional Halton sequence, and the star discrepancy of the sequence generated by the i-th processor satisfies $D_N^* < c_s \frac{(\ln N)^s}{N} + O((\ln N)^{s-1}/N)$, where $c_s = \frac{1}{s!} \prod_{j=0}^{s-1} \frac{p_{i+jm} - 1}{\ln p_{i+jm}}$ (see [1]).

We now discuss the error analysis of the estimates obtained by our parallel scheme, when processors run at equal and different speeds. As before, the computing environment has m processors, and N is the total number of samples generated by all of the processors at a certain time. Let us first consider the case when the processors produce vectors at different speeds. We assume that each processor has a fixed speed relative to the other processors. Let $\lambda_i(N)$ be the function that gives the number of vectors produced by the processor i, when the total number of vectors generated by all processors is N. Clearly, $N = \sum_{i=1}^{m} \lambda_i(N)$. From our assumption that a processor produces numbers at the same relative (physical) speed with respect to the other processors, we conclude that

$$\lim_{N \to \infty} \frac{\lambda_i(N)}{N} = \lambda_i, \tag{9}$$

for some constant λ_i. This constant can be thought of as the limiting proportion of vectors produced by the ith processor to the total number of vectors.

If we label the N vectors generated by the processors as $z^{(1)}, z^{(2)}, ..., z^{(N)}$, then the final estimate of our parallel scheme is obtained by evaluating the function values $f(z^{(n)})$, $n = 1, ..., N$, and computing their average $e = \frac{1}{N} \sum_{n=1}^{N} f(z^{(n)})$. Computing e in this way is especially useful if there is a single processor that collects vectors $z^{(n)}$ as they are produced, and then computes the function values and averages. In this way, the processor that collects the information does not need to keep track of where each vector was produced. Such a scheme would be helpful when point generation is relatively more expensive than function evaluation. In most practical situations, however, function evaluation is much more expensive than point generation. Therefore we can let each processor compute the function values at the vectors it produces, and then compute the average of the function values. We obtain m individual estimates $e_i = \frac{1}{\lambda_i(N)} \sum_{n=1}^{\lambda_i(N)} f(q_i^{(n)})$, $i = 1, ..., m$. Here $q_i^{(n)}$ refers to the (finite or infinite dimensional) vector $\left(\phi_{p_i}(n), \phi_{p_{i+m}}(n), ...\right)$ of (8). We observe that

$$e = \frac{1}{N} \sum_{n=1}^{N} f(z^{(n)}) = \sum_{i=1}^{m} \frac{\lambda_i(N)}{N} \frac{1}{\lambda_i(N)} \sum_{n=1}^{\lambda_i(N)} f(q_i^{(n)}) = \sum_{i=1}^{m} \frac{\lambda_i(N)}{N} e_i,$$

which shows us how to compute the grand estimate e from the individual estimates e_i in this second approach. We now use the Koksma-Hlawka inequality to derive the error bound

$$\left| e - \int_{I^s} f(x)dx \right| \leq \sum_{i=1}^{m} \frac{\lambda_i(N)}{N} \left| e_i - \int_{I^s} f(x)dx \right| \leq \sum_{i=1}^{m} \frac{\lambda_i(N)}{N} D^*_{\lambda_i(N)}(q_i^{(n)}) V(f),$$

for functions of bounded variation in the sense of Hardy and Krause. We note that our parallel quasi-Monte Carlo scheme is not a point-set method, and taking limits as $N \to \infty$ in the above inequality, we obtain $\left| e - \int_{I^s} f(x)dx \right| \to 0$, since $\frac{\lambda_i(N)}{N} \to \lambda_i$ and $D^*_{\lambda_i(N)}(q_i^{(n)}) \to 0$.

We now consider the special case when all processors run at equal speeds. Then $\lambda_i(N) = N/m$, $e = (e_1 + ... + e_m)/m$, and the above inequality simplifies to the following

$$\left| e - \int_{I^s} f(x)dx \right| \leq \frac{1}{m} \sum_{i=1}^{m} D^*_{N/m}(q_i^{(n)}) V(f). \tag{10}$$

The above inequalities show that our parallel scheme produces estimates that converge with the same asymptotic bound of a low-discrepancy sequence. However they fall short of explaining if (and why) the grand estimate e is better than a typical estimate e_i. It is possible to say more about the accuracy of e by randomly assigning Halton sequences to processors, instead of assigning a fixed sequence to each processor. This simple randomization makes e an asymptotically unbiased estimator for $\int_{I^s} f(x)dx$, with variance $O(\frac{(\log N/m)^{2s}}{N^2/m})$. Details of this randomization approach can be found in [14].

Another randomization approach would be to assign randomized low-discrepancy sequences such as scrambled low-discrepancy sequences [15], randomly shifted low-discrepancy sequences [24], or random-start Halton sequences [25], to each processor. Then e becomes an unbiased estimator for

$\int_{I^s} f(x)dx$ with[4]

$$Var\left[e - \int_{I^s} f(x)dx\right] = E\left|e - \int_{I^s} f(x)dx\right|^2$$

$$= \sum_{i=1}^{m} \left(\frac{\lambda_i(N)}{N}\right)^2 E\left|e_i - \int_{I^s} f(x)dx\right|^2 \leq (V(f))^2 \sum_{i=1}^{m} \left(\frac{\lambda_i(N)}{N}\right)^2 \left(D^*_{\lambda_i(N)}\right)^2,$$

which, for equal speeds, simplifies to

$$Var\left[e - \int_{I^s} f(x)dx\right] \leq \left(D^*_{N/m}\right)^2 (V(f))^2 /m.$$

We note that the variance in both the above equations (equal speeds and the general case) is $O(m \log^{2s} N/N^2)$. Thus both equations yield the same asymptotic order.

3.3 Scrambled Halton Sequences

Several authors observed that the uniformity properties of the Halton sequence can be improved significantly by carefully modifying the coefficients $a_j(n)$ in the expansion of $\phi_{b_i}(n)$ in (7). Braaten and Weller [2], Chaix and Faure [4], and Faure [11] all introduced such methods that decrease some "form" of discrepancy of the Halton sequence. Here we will consider the approach taken by Faure in [11], where he introduces permutations σ on the digit set $\{0, ..., b_i - 1\}$, and modifies the definition of $\phi_{b_i}(n)$ as $\phi_{b_i}(n) = \sum_{j=0}^{\infty} \sigma(a_j(n)) b_i^{-j-1}$. He gives a systematic construction of permutations σ for any given base b_i that minimize the extreme discrepancy of the one-dimensional van der Corput sequences $\{\phi_{b_i}(n)\}$. When these permuted one-dimensional sequences are used to construct $q^{(n)}$ from (6), we obtain the so-called scrambled Halton sequence. Therefore, by construction, the scrambled Halton sequence has one-dimensional projections with improved uniformity properties. This feature makes the scrambled Halton sequence especially attractive in problems where most of the variation is concentrated in lower-dimensional hypercubes.

In Sect. 4.1, we will use Halton and scrambled Halton sequences in our parallelization scheme as described in Sect. 3, and apply them to a problem from computational finance. The numerical results will be seen to be in favor of the scrambled Halton sequence, and therefore we will use only scrambled Halton sequences in the subsequent numerical experiments.

4 Numerical Examples

4.1 Computational Finance

Here we consider a problem from computational finance: pricing of geometric Asian options. The exercising features of the option are assumed to be

[4] This was pointed out to us by Fred J. Hickernell.

European. The price of the underlying asset, $S = S(t)$, is assumed to satisfy the Ito process $dS = \mu S dt + \sigma S dX$, where t is time, μ and σ are the drift and volatility of the underlying respectively, and $X = X(t)$ is a standard Brownian motion. For a finite sequence of asset prices $S_0, S_1, ..., S_K$, the payoff function is $h(S_0, S_1, ..., S_K) = \max(G(S_0, S_1, ..., S_K) - E, 0)$, where $G(S_0, S_1, ..., S_K) = \left(\prod_{i=0}^{K} S_i\right)^{1/(K+1)}$ is the geometric average of the stock prices, and E is the strike price. The option value at the present time is the expected value $\mathbf{E}\left[e^{-rT} h(S_0, S_1, ..., S_K)\right]$. In Monte Carlo simulation, we approximate this expected value by generating N random walks $w_j = S_0, S_1^j, ..., S_K^j$, $j = 1, ..., N$ and evaluating $\frac{1}{N}\sum_{j=1}^{N} e^{-rT} h(w_j)$.

Geometric Asian options have analytic solutions in Black-Scholes model. This enables us to compare the actual error produced by our simulation techniques. Details of this problem including formulas to compute its exact Black-Scholes value, as well as the way we discretize the Ito process, can be found in [5]. In the following numerical results, we take $S_0 = E = 100$, $\sigma = 0.2$, risk free rate $r = 0.06$, continuous dividend rate $\rho = 0.03$, $K = 100, T = 1$, with $\mu = r - \rho$. The exact Black-Scholes value of the option is 4.97671294245.

Figure 1 compares the results when Halton and scrambled Halton sequences are used in our parallelization scheme, which we will refer to as the *parameterized sequence* approach. It is clear that the scrambled Halton sequence implementation is superior to the Halton sequence implementation.

We next compare our approach with the sequences used in Schmid and Uhl [16,17], which were Niederreiter sequences with leap-frog and blocking parallelization. Figure 2 shows the results on two processors and Fig. 3 on eight processors, with equal processor speeds. The parameterized sequence approach is clearly superior in the two processor case, although the advantages diminish in the eight processor case.

We next present results with unequal processor speeds. We can see from Figs. 4 and 5 that our approach outperforms leap-frogged and blocked Niederreiter sequences significantly, especially the former.

We can observe from the above figures that the blocked sequence significantly outperforms the leap-frogged sequence with unequal speeds, while the parameterized sequence approach outperforms both of these approaches in equal and unequal speeds. However, the performance of the blocked scheme is surprisingly good, as also noted in [16,17]. On the other hand, if blocks overlap, then the blocking scheme yields much worse results, as can be seen from Fig. 6, where we choose small block sizes to demonstrate defects due to overlap of blocks.

We wish to note, however, that blocking with the scrambled Halton sequence (without overlap) performed quite well, as shown in Fig. 7. The leap frog scheme performed well with equal speeds, but poorly with unequal speeds (the latter figure is not shown). As with the Niederreiter sequence, the blocking sequence too performs worse in the presence of overlap, as expected. This

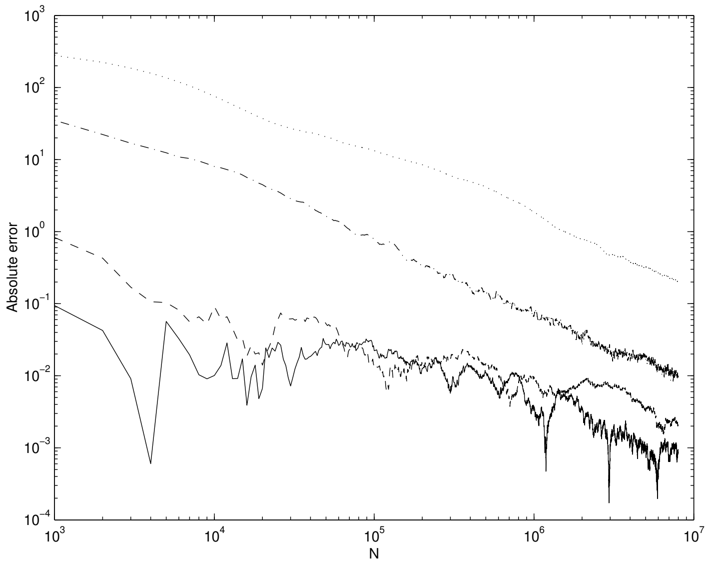


Fig. 1. Plot of absolute error versus number of simulations for Halton (dotted line: 8 processors, dash-dot:2 processors) and scrambled Halton (dashed: 8, solid: 2 processors) parallelized using parameterized sequences.

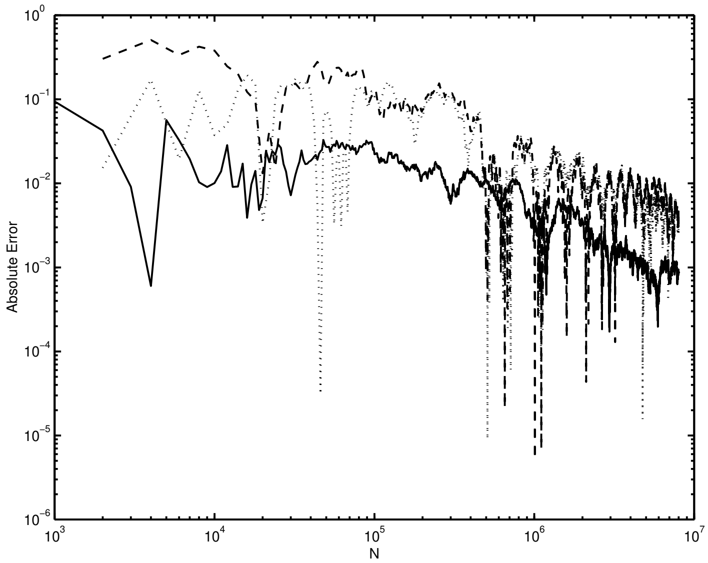


Fig. 2. Plot of absolute error versus number of simulations for Niederreiter and scrambled Halton schemes on 2 processors with equal speeds. The former used leap-frog (dotted line) and blocking (dashed) with block size $= 8 \times 10^7$, while the latter used parameterized sequences (solid).

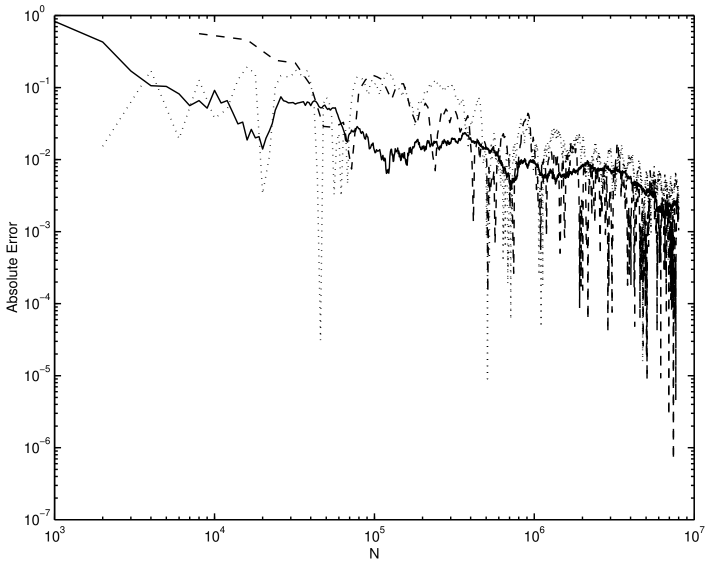


Fig. 3. Plot of absolute error versus number of simulations for Niederreiter and scrambled Halton schemes on 8 processors with equal speeds. The former used leap-frog (dotted line) and blocking (dashed) with block size = 8×10^7, while the latter used parameterized sequences (solid).

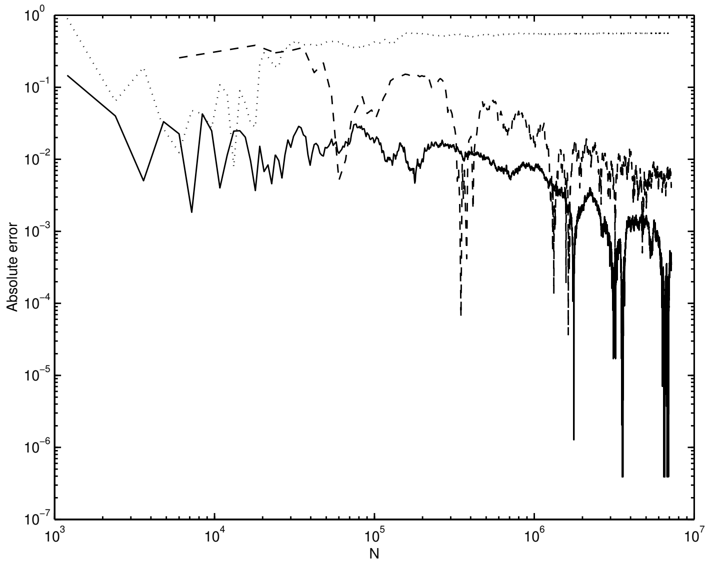


Fig. 4. Plot of absolute error versus number of simulations for Niederreiter and scrambled Halton schemes on 2 processors with unequal speeds (ratio: 5:1). The former used leap-frog (dotted line) and blocking (dashed) with block size = 8×10^7, while the latter used parameterized sequences (solid).

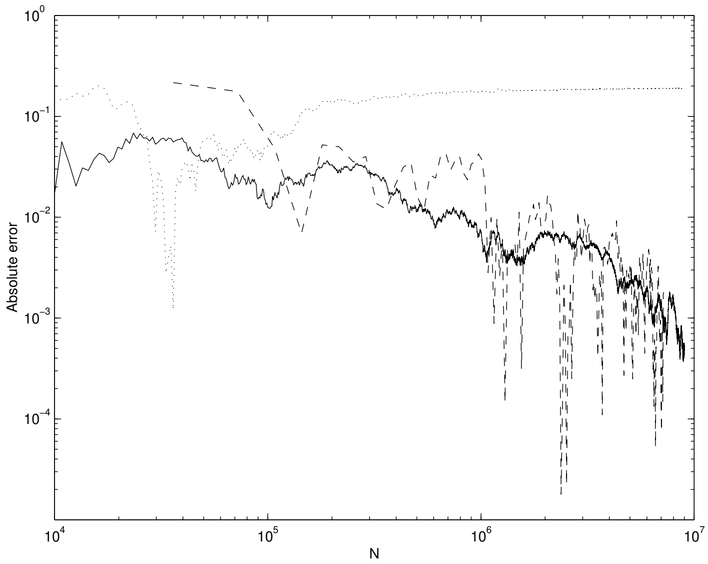


Fig. 5. Plot of absolute error versus number of simulations for Niederreiter and scrambled Halton schemes on 8 processors with unequal speeds (ratio: 8:7:6:5:4:3:2:1). The former used leap-frog (dotted line) and blocking (dashed) with block size $= 8 \times 10^7$, while the latter used parameterized sequences (solid).

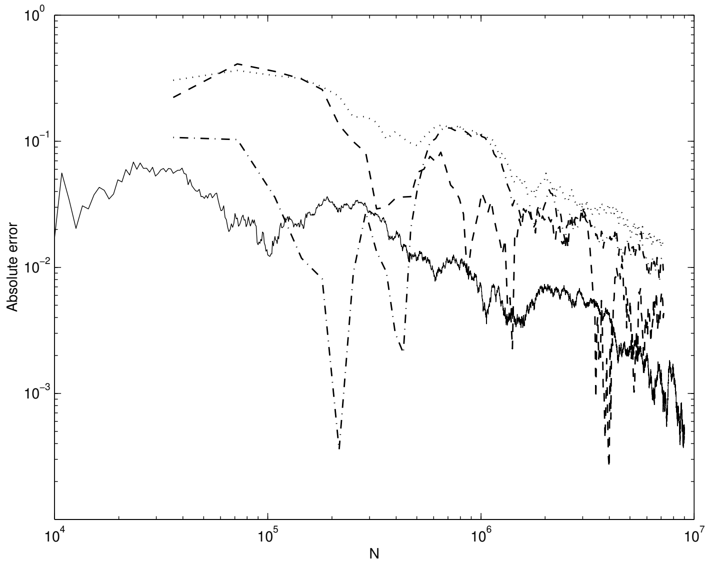


Fig. 6. Plot of absolute error versus number of simulations for blocked Niederreiter and scrambled Halton schemes (solid line) on 8 processors with unequal speeds (ratio: 8:7:6:5:4:3:2:1). The former used blocking (dashed) with block sizes of 10^3 (dotted), 10^4 (dot-dash), and 10^5 (dash).

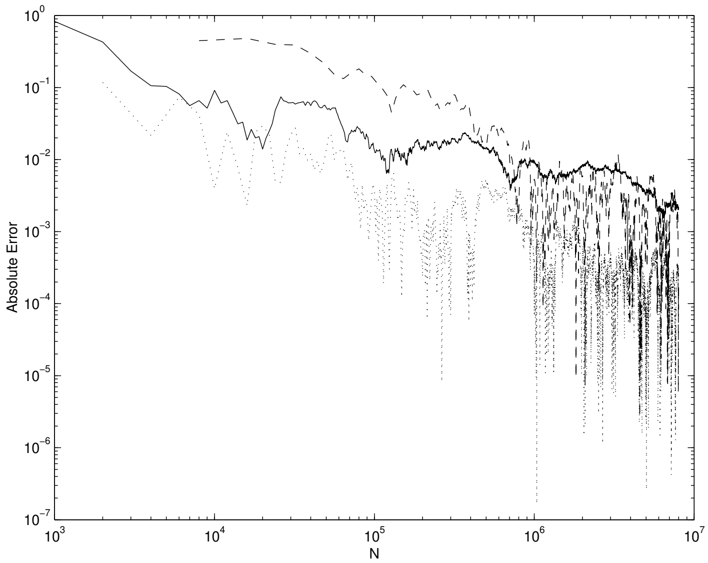


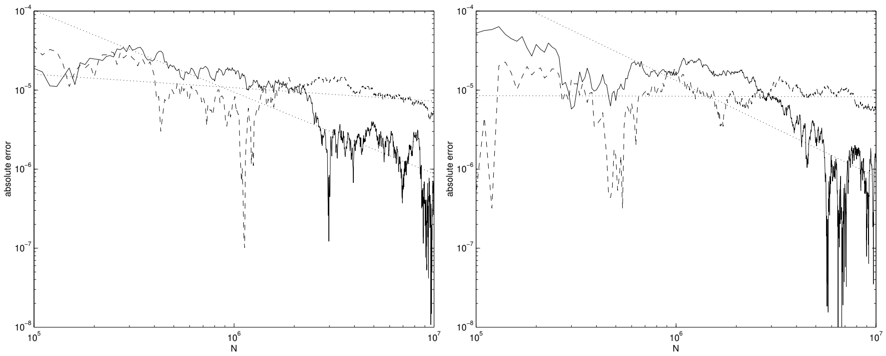
Fig. 7. Plot of absolute error versus number of simulations for parallel scrambled Halton schemes on 8 processors with equal speeds. The plot shows leap-frog (dotted line), blocking (dashed) with block size = 8×10^7, and parameterized sequences (solid).

can be a limitation in real applications, where users may continue a previous simulation, resulting in the block size being exceeded. Yet another important disadvantage of the blocking scheme is that, unlike with the parameterized sequence approach, its theoretical properties are not apparent especially when we allow overlaps and unequal speeds.

4.2 Particle Transport Theory

We now consider a simple deep penetration type problem from particle transport theory. It was used by Spanier [21] as a model problem to test new simulation ideas. In this problem, neutrons that are emitted by a nuclear source enter a slab of thickness d from its left face. In the slab, a neutron can either be absorbed, back scattered or forward scattered upon a collision. The distance between consecutive collisions of a neutron is a random variable with the probability density function[5] $f(x) = e^{-x}$. The problem is to find the probability of transmission p_t, i.e, the probability that a neutron that enters the slab from its left face, exits the slab from its right face. We will estimate p_t by Monte Carlo simulation. There is an analytic solution to this

[5] In conventional transport applications, the inter-collision probability density function is $f_c(x) = ce^{-cx}$, where c is a function of the material properties of the block. Here we have normalized the problem by choosing $c \equiv 1$.



(a) (b)

Fig. 8. Plot of absolute error versus number of simulations on 2 (fig a) and 8 (fig b) processors with equal speeds. The dashed line is the PRNG and the solid line is scrambled Halton with parameterized sequences. The dotted line along each curve is the corresponding regression plot.

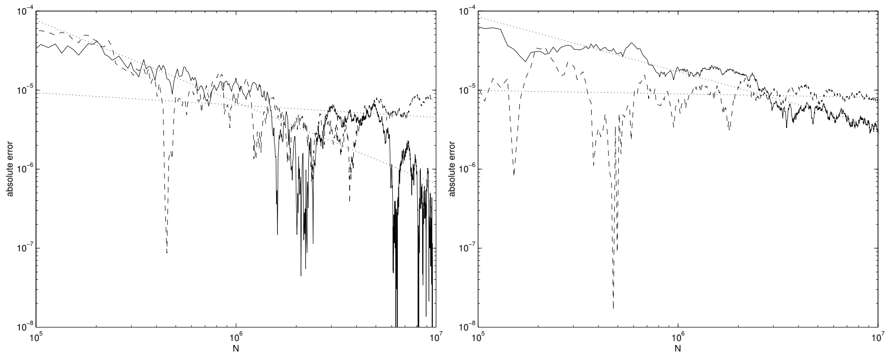
problem (see [20]), so we are able to compare different simulation methods by the actual error they produce.

In the following results, we let the probability of scattering per collision, p_s, be 0.9, and the thickness, d, be 20 units. The events back and forward scattering are assumed to be equally likely. The exact solution is $p_t = 1.31 \times 10^{-3}$. We note that this problem is an infinite dimensional problem: there is no upper bound for the number of steps a random walk takes. In this problem, we compare our scrambled Halton approach with a parallel pseudorandom number generator (a modified additive lagged Fibonacci generator with lags (1279, 861) from SPRNG [13].

Figure 8 compares SPRNG with our approach with equal speeds, while Fig. 9 compares them with unequal speeds. In both cases, it is clear that our approach outperforms SPRNG for large N. We fit a regression line to match the data on the log-log plot and observed that the error with the quasi-random sequence ranged from $O(N^{-1.2})$ to $O(N^{-0.69})$, while the error with the pseudo-random sequence ranged from $O(N^{-0.17})$ to $O(N^{-0.044})$ (with SPRNG's error being much worse than theoretically expected). (Note that the regression data were at equally spaced values of N, and therefore on a log-log scale, there is relatively more data for larger values of log N.)

5 Conclusions

We have proposed a parallelization scheme based on the idea of assigning different low-discrepancy sequences to different processors of the parallel computing environment. Our choice for the underlying low-discrepancy sequences was the family of scrambled Halton sequences, making our scheme



(a) (b)

Fig. 9. Plot of absolute error versus number of simulations on 2 (fig a) and 8 (fig b) processors with unequal speeds (ratios 5:1 and 8:7:6:5:4:3:2:1 respectively). The dashed line is the PRNG and the solid line is scrambled Halton with parameterized sequences. The dotted line along each curve is the corresponding regression plot.

applicable to finite as well as infinite dimensional problems. The numerical results show the advantages of our parallelization scheme over the conventional leap-frogging and blocking ideas especially when we consider situations where the processors run at different speeds, and users make poor decisions on block sizes causing overlaps. Clearly such scenarios are likely to happen in distributed computing environments. We also note the simple theoretical analysis of our scheme.

Acknowledgement

We thank Fred J. Hickernell and the anonymous referee for their valuable comments and suggestions.

References

1. Atanassov, E.I.: On the discrepancy of the Halton sequences. Technical Report. Bulgarian Academy of Sciences, 1999.
2. Braaten, E., Weller, G.: An Improved Low-Discrepancy Sequence for Multidimensional Quasi-Monte Carlo Integration. J. Comp. Physics **33** (1979) 249-258
3. Bromley, B.C.: Quasirandom number generators for parallel Monte Carlo algorithms. Journal of Parallel and Distributed Computing, **384** (1996) 101–104
4. Chaix, H., Faure, H.: Discrepance et diaphonie en dimension un. Acta Arithmetica **LXIII** (1993) 103-141
5. Clewlow L., Strickland C.: Implementing Derivatives Models. John Wiley & Sons, 1998
6. Coddington, P.: Random number generators for parallel computers, 1997. http://nhse.cs.rice.edu/NHSEreview/96-2.html.

7. Coddington, P.: Random number generators for parallel computers, 28 April 1997. www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps.
8. De'Matteis, A., Pagnutti, S.: Parallelization of random number generators and long-range correlations. Parallel Computing, **15** (1990) 155–164
9. De'Matteis, A., Pagnutti, S.: Long-range correlations in linear and non-linear random number generators. Parallel Computing, **14** (1990) 207–210
10. De Doncker, E., Zanny, R., Ciobanu, M., Guan, Y.: Distributed quasi Monte-Carlo methods in a heterogeneous environment. Proceedings of the Heterogeneous Computing Workshop 2000, (2000) 200–206
11. Faure, H.: Good Permutations for Extreme Discrepancy. Journal of Number Theory **42** (1992) 47–56
12. Li, J.X., Mullen, G.L.: Parallel computing of a quasi-Monte Carlo algorithm for valuing derivatives. Parallel Computing, **26** (2000) 641–653
13. Mascagni, M., Srinivasan, A.: SPRNG: a scalable library for pseudorandom number generation, 2001. ACM Transactions on Mathematical Software, **26** (2000) 436–461
14. Ökten, G.: Error Estimation for Quasi-Monte Carlo Methods. Monte Carlo and Quasi-Monte Carlo Methods *1996 (*H. Niederreiter et al., eds.*),* Lecture Notes in Statistics, Vol. 127, pp. *353-368,* Springer, New York, 1998
15. Owen, A. B.: Randomly Permuted (t,m,s)-Nets and (t,s)-Sequences. Monte Carlo and Quasi-Monte Carlo in Scientific Computing (H. Niederreiter and P.J.-S. Shiue, eds.), Lecture Notes in Statistics, Vol. 106, pp. 121-148, Springer, New York, 1995
16. Schmid, W., Uhl, A.: Parallel Quasi-Monte Carlo integration using (t,s)-sequences. Lecture Notes in Computer Science, **1557** (1999) 96–106
17. Schmid, W., Uhl, A.: Techniques of parallel Quasi-Monte Carlo integration with digital sequences and associated problems. Mathematics and Computers in Simulation, **1875** (2000) 1–9
18. Sobol'. I.M.: On the evaluation of infinite-dimensional integrals. USSR Comp. Math Phys **1** (1961) 1086-1091
19. Sobol', I.M.: Increasing the Efficiency of Statistical Sampling with the Aid of Infinite-Dimensional Uniformly Distributed Sequences. Proc. First World Congress Bernoulli Soc. 2, pp.743-746, VNU Sci. Press, Utrecht, 1987
20. Spanier, J.: An Analytic Approach to Variance Reduction. SIAM J. Appl. Math. **18** (1970) 172-190
21. Spanier, J.: Quasi-Monte Carlo methods for particle transport problems. Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (H. Niederreiter and P.J.-S. Shiue, eds.), Lecture Notes in Statistics, Vol. 106, pp. 121-148, Springer, New York, 1995
22. SPRNG – Scalable Parallel Random Number Generators. http://www.ncsa.uiuc.edu/Apps/SPRNG.
23. Srinivasan, A., Ceperley, D.M., Mascagni, M.: Random number generators for parallel applications. In Ferguson, D., Siepmann, J. I., Truhlar, D. G. editors, *Monte Carlo Methods in Chemical Physics*, volume 105 of *Advances in Chemical Physics*, pages 13–36. John Wiley and Sons, New York, 1999
24. Tuffin, B.: On the use of low discrepancy sequences in Monte Carlo methods. Monte Carlo Methods and Applications **2** (1996) 295-320
25. Wang, X., Hickernell, F. J.: Randomized Halton Sequences. Mathematical and Computer Modelling **32** (2001) 887-899