Parallel and Distributed Computing Issues in Pricing Financial Derivatives through Quasi Monte Carlo

Ashok Srinivasan
Department of Computer Science, Florida State University,
Tallahassee FL 32306, USA, Email: asriniva@cs.fsu.edu

Abstract

Monte Carlo (MC) techniques are often used to price complex financial derivatives. The computational effort can be substantial when high accuracy is required. However, MC computations are latency tolerant, and are thus easy parallelize even with high communication overheads, such as in a distributed computing environment. A drawback of MC is its relatively slow convergence rate, which can be overcome through the use of Quasi Monte Carlo (QMC) techniques, which use low discrepancy sequences. We discuss the issues that arise in parallelizing QMC, especially in a heterogeneous computing environment, and present results of empirical studies on arithmetic Asian options, using three parallel QMC techniques that have recently been proposed. We expect the conclusions to be valid for other applications too.

1. Introduction

A financial derivative is a function of some basic variables, such as stock price [4]. Derivatives are an important mechanism to control financial risk, and the volume of trade in derivatives is enormous. The price of a derivative is determined by first choosing a suitable mathematical model, and then solving it. Only the simplest models have analytical solutions, and thus numerical techniques are often required. We will use the "Arithmetic Asian option", to be defined in section 2, as an example throughout this paper, to illustrate the issues that arise in parallelization.

Monte Carlo (MC) techniques are often used to solve complex options. If high accuracy is desired, then the computational effort can be tremendous for complex models, and thus there has been recent interest in parallelization [7, 9, 13], both with MC and non-MC techniques. MC techniques can be parallelized by executing identical sequential algorithms on each processor, independently, with just the pseudo-random number sequence being different, and the results from each processor finally being combined. Though implementation details require that we have occasional communication, the speed-up is essentially linear, leading to the common perception that MC is "embarrassingly parallel". However, popular techniques of parallelizing pseudorandom number generators (PRNG) can lead to a situation where the efficiency in terms of time is 1, however, the efficiency in terms of error-reduction is much lower. In fact, the results can even be wrong, as demonstrated in [12] and several other studies.

An important drawback of MC is that its convergence rate of $O(1/\sqrt{N})$ with N samples, is not sufficiently good for many applications. This can be overcome through the use of Quasi Monte Carlo (QMC) techniques, which use low discrepancy sequences (LDS). Unlike pseudo-random sequences used in MC, which try to produce a sequence with the statistical properties of a random sequence, LDSs try to ensure uniformity, rather than randomness. A deterministic error bound of $O(\log^M N/N)$ in M dimensional integration can be obtained from the Koksma-Hlawka inequality. When the effective dimensionality of the problem is not very high, QMC can converge faster than MC, and several studies, such as [5], have demonstrated its advantages over MC in a variety of financial applications. The issues that arise in parallelizing QMC are quite different from those that arise in parallelizing MC, primarily due to the fact that low discrepancy sequences are inherently deterministic, and there has been interest in recent times in parallelizing QMC [1, 9, 10, 11]. We will discuss the issues that arise in parallel QMC, and three techniques that have recently been proposed for it, in section 3.

We demonstrate the different issues that arise in QMC parallelization, by performing empirical tests on the Arithmetic Asian option, using three different QMC parallelization techniques under different scenarios. The results are presented in section 4. The different techniques have different characteristics, and we conclude by summarizing them, and potential improvements to the techniques, in section 5.

2. Arithmetic Asian option

We now give a simplified abstraction of options, then define the option to be studied, and finally outline the numerical scheme.

An option is a class of derivatives that gives its holder the right to something, but not an obligation. The holder needs to pay something for obtaining this right, which is the price of the option. We now describe the Arithmetic Asian call option [2, 4], which we use to demonstrate the issues that arise in parallelizing QMC. This option gives its holder a payoff defined by:

$$\max(0, S_{am} - X), \tag{1}$$

where X, called the "strike price", is a quantity decided in advance, and S_{am} is the arithmetic mean of the price of the underlying asset, observed at predetermined points in time $t_1, t_2, ..., t_M$. Thus $S_{am} = (1/M)\sum_{i=1}^{M} S_{t_i}$, where S_{t_i} is the price of the underlying asset at time t_i. We take the exercising feature to be "European", implying that the right can be exercised only at a predetermined expiration time T.

In order to determine the price of the option, we model the underlying asset as following geometric Brownian motion defined by the following stochastic differential equation [2]:

$$dS = \mu S dt + \sigma S dz \tag{2}$$

where μ and σ are given constants, and dz is normally distributed with mean 0 and variance dt. Knowing the initial value S_0 of the underlying asset, we can perform a MC simulation to estimate the values of S_{t_i} and subsequently estimate the payoff, and from that the price.

We now provide further details on the simulation. In order to avoid discretization errors, we simulate not S directly from Equation (2), but rather, its logarithm [2], given by:

$$\Delta x = \nu \Delta t + \sigma \Delta z \tag{3}$$

where $x = \ln S$, $\nu = r - \delta - \frac{\sigma^2}{2}$, and Δz is normally distributed with mean 0 and variance Δt. Here, r and δ are given constants, and $\mu = r - \delta$. We consider the case where $t_i = i\Delta t,\ i = 1, ..., M,\ T = M\Delta t$.

Given the known initial value of the stock price S_0, we take M samples from the normal distribution, compute x_{t_i} using Equation (3), and compute $S_{t_i} = \exp(x_{t_i})$ for each i. This enables us to estimate S_{am}, from which we estimate the price as $\exp(-rT)\max(0, S_{am} - X)$ [2]. Several such simulations are performed to get an accurate estimate for the price.

The primary difference between MC and QMC is that MC generates samples for Δz by choosing *random* points from the uniform distribution and transforming them to a normal variate, while QMC chooses *uniformly* distributed points in M dimensional space, and transforms them to normal variates (we use inverse transform to accomplish this). Each component of the transformed point is used as a sample for Δz. Each simulation uses one M-dimensional point, and the (Q)MC simulation is the estimation of an M-dimensional integral.

3. Quasi Monte Carlo parallelization issues

While there has been much work done on the issue of parallel PRNGs (PPRNG), there have been relatively fewer studies on the issues that arise in parallelizing LDSs. 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.

The parallelization issues with QMC are different from those with MC; if we choose n_i points on processor $i,\ 0 \le i \le P-1$, then we wish the "discrepancy" (which is a measure of the uniformity) of these points to be similar to that of a single low-discrepancy sequence of $\sum_{i=0}^{P-1} n_i$ points. In an unreliable system, we should allow for processors to fail, and therefore the discrepancy of the points on each processor must be small, and the discrepancy of points on any subset of processors must be small too. In contrast, with a PPRNG, we wish to avoid correlations between and within the random number sequences.

Three schemes have been proposed in recent times [1, 7, 9, 10, 11] for parallelizing low-discrepancy sequences.

1. *Leap frog:* Let $x^0, x^1, x^2, ...$ be an M-dimensional low-discrepancy sequence (each element x^k is an M-dimensional vector). We assign the sequence $x^i, x^{i+P}, x^{i+2P}, ...$ to processor $i,\ 0 \le i < P$.

2. *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.

3. *Parameterization:* Parameterization was introduced for the Halton and Scrambled Halton sequences in [9]. In a sequential computation, the Halton sequence is defined by $x^k = (\phi_{b_0}(k), ..., \phi_{b_{M-1}}(k))$ where b_is are primes, and ϕ_{b_i} is the radical-inverse function in base b_i, defined by $\phi_{b_i}(k) = \sum_{j=0}^{\infty} a_j(k) b_i^{-j-1}$, $k = \sum_{j=0}^{\infty} a_j(k) b_i^j$. If we let p_i denote the i th prime, then usually b_i is taken to be p_i. In the parallelization proposed in [9], the i th processor uses the sequence de-

fined by $x^k = (\phi_{p_i}(k), \phi_{p_{i+P}}(k), ..., \phi_{p_{i+(M-1)P}}(k))$. This technique can also be directly applied to the "scrambled" versions of the sequence, which have better uniformity properties.

Leap frog and blocking schemes require the ability to "jump ahead" in a sequence, in order to have efficient implementations. Such facilities were implemented for the Sobol' sequence by Bromley [1] and for the (t, s) sequences in general by Schmid and Uhl [10]. In related work, Li and Mullen [7] introduced a leap-frog scheme for (t, m, s) nets and used it to solve financial derivative problems. We next discuss some properties of the above three schemes.

1. 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/P 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, $B = N/P$. 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 $N/P > B$. Thus results of parallel quasi-Monte Carlo with blocking scheme will typically not produce a result identical to that of a sequential run. The parameterized sequence will not produce a result identical to that of a sequential calculation, when run on more than one processor.

2. If processors run at different speeds, then even the leap frog scheme will not give a result identical to that of a sequential computation. (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 computation.)

3. If processors run at different speeds and we use practical synchronization strategies, then the results obtained by leap frog and blocking will be identical to a sequential computation that uses the original sequence with some "gaps". However, for a general low-discrepancy sequence, uniformity 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 [11]. In contrast, theoretical results can be given for parameterization, justifying convergence of the results even when processors run at different speeds [9], though the results are not as strong as desired. Furthermore, while the asymptotic discrepancy is good, it is not clear that it is good for the number of points used in practice.

4. Empirical tests

We wish to determine the effectiveness of the different QMC parallelization strategies for the derivative pricing application. We first mention some related work, and next state the goals of our tests in section 4.1. We describe the test methodology in section 4.2, and present the results in section 4.3.

Related work: Schmid and Uhl [10, 11] have performed empirical tests on certain integrals, comparing leap frog and blocking schemes with (t, s) sequences. The three techniques are compared empirically in [9], including tests on a geometric Asian option (for which an analytical solution is available). However, the tests primarily used (t, s) sequences for leap frog and blocking, but used scrambled Halton for parameterization. We wish to apply the same sequence, namely, scrambled Halton, to all the three techniques, so that the effect of the parallelization techniques can be distinguished from those of the sequence. Furthermore, the largest number of processors in the above study was 8, while we extend it to 16 processors.

4.1. Test goals

1. We want to compare the effectiveness of the three strategies when the processor speeds are equal.

2. We want to test their relative performance when the processors speeds are unequal (in a heterogeneous system).

3. We wish to study the scalability of these techniques with the number of processors and with increase in dimensionality (number of time steps, here) of the problem. Though the maximum number of processors we consider is not very large, it is sufficient to distinguish the behaviors of the three techniques.

4.2. Test methodology

- We took the following parameters for the options, as in Clewlow [2]: strike price $X = 100$, initial price of the asset $S_0 = 100$, expiration time $T = 1$, volatility $\sigma = 0.2$, risk free rate $R = 0.06$, and continuous dividend rate $\delta = 0.03$. In high dimensional simulations, we took the number of time steps $M = 40$, while in low dimensional simulations we took $M = 10$. The number of processors used varied amongst 1, 4, 8, and 16.

- The exact solution to this problem is not known analytically. In order to determine an "exact" solution, we performed 10^9 simulations each two different PPRNGs, (i) a modified additive lagged Fibonacci Generator from the `SPRNG` [8] package and (ii) a 48-bit Linear Congruential Generator from the same package. The estimate and standard error using each generator was computed to confirm that their answers were sufficiently close, to rule out the possibility that the result was biased due to defects in the PPRNGs, and then the results were averaged together to compute the "exact" solution. From the standard deviation, we determined that this solution is accurate to around 5 digits. The exact solution with $M = 40$ was 5.22897, while with $M = 10$ it was 5.53309.

- Normally, variance reduction techniques can be used to get more accurate results. However, since the purpose of the study is to compare the different techniques, we did not use such techniques, so that defects in the methods would become apparent.

- The parallel QRNGs we used are not implemented optimally, and therefore we plot error against the number of simulations, rather than against time. Efficient implementations would consume about the same amount of time for each method considered, and thus using the number of points is a fairer comparison. Furthermore, this enables us to simulate computations for a larger number of processors on fewer.

- We primarily use the scrambled Halton sequence, with the scrambling due to Faure [3] in most experiments. This sequence, unlike the plain Halton sequence, has been shown to be extremely effective in several calculations [9]. Furthermore, it can be effectively parallelized in all the three techniques.

4.3. Test results

We compare the three techniques in the 40 dimensional ($M = 40$) case, with equal processor speeds, in Figure 1.

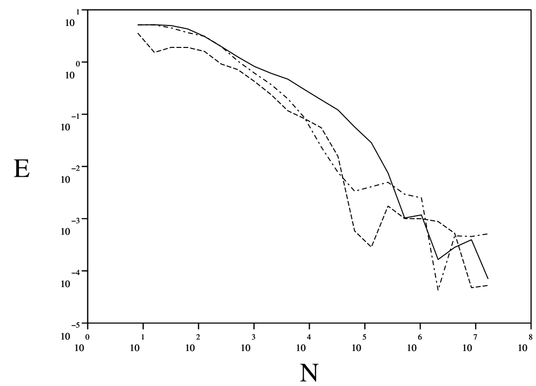

Figure 1. Plot of absolute error (E) vs number of simulations (N) for the Arithmetic Asian option (time steps = 40), using scrambled Halton on 4 processors – equal speeds, parallelized with: leap-frog (dash-dotted line), blocking (dashed) – block size = 10^8, and parameterization (solid).

All the techniques perform similarly. The good performance of leap frog agrees with the observations of [9], but is contrary to those in [10].

We next present results with unequal processor speeds. We can see from Figure 2 that blocking and parameterization are about equally good, while leap frog is significantly worse. We observed from our data for the equal speed case that the leap frog results from each processor are poor. However, when combined, they gave good results. With unequal speeds, different processors get different weights, and those errors may not offset each other sufficiently. Thus leap frog ought to be avoided in a heterogeneous computation, which may often arise in a distributed computational environment. However, it may be feasible to modify leap frog to give better results on each processor. Instead of leaping by P, we may leap by a different amount, for example, using some leaps suggested in [6]. However, the leaps suggested in [6] improve the quality of a single Halton sequence; the consequences to the discrepancy of the parallel sequence is not clear.

We next wish to study the scalability of the approaches. We can observe from Figures 1 and 3 that parameterization performs poorly when the number of processors is larger. However, the error still decreases, unlike in the failure of leap frog with unequal speeds. This can be explained by re-formulating parameterization as follows: consider the MP dimensional sequence $y^0, y^1, y^2, ...$, where $y^i = (y^i_0, y^i_1, ..., y^i_{MP-1})$. In the parameterization technique, as it has been proposed, processor j

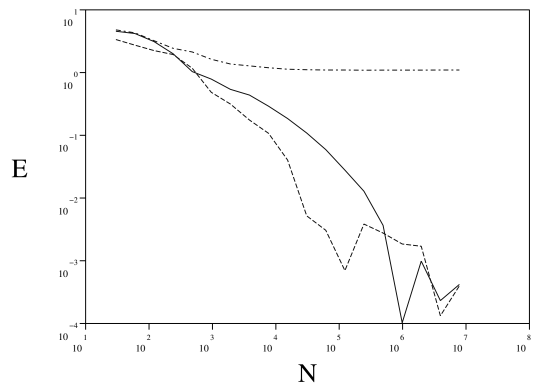

Figure 2. Plot of absolute error (E) vs number of simulations (N) for the Arithmetic Asian option (time steps = 40), using scrambled Halton on 4 processors – speed ratios 8:4:2:1, parallelized with: leap-frog (dash-dotted line), blocking (dashed) – block size = 10^8, and parameterization (solid).

gets the sequence $x(j)^0, x(j)^1, x(j)^2, \cdots$, where $x(j)^i = (y^i_j, y^i_{j+P}, \cdots, y_{j+(M-1)P})$. If the y^is are uniformly distributed, then the projection on a subset of coordinates too is uniformly distributed, and so the sequence on each processor too will have a low discrepancy. This justifies the asymptotic properties of parameterization, and explains why the error keeps decreasing.

A limitation of the above argument is that many low discrepancy sequences behave poorly in high dimensions for realistic numbers of points. When the number of processors gets larger, the quality of the sequences could get poorer, since the dimension of y^is is a function of P. However, it may be feasible to modify the scheme to alleviate this problem, by taking different projections, rather than the one above. We discuss this further later.

We also need to consider scalability with dimensionality of the problem. We can compare Figure 3 with a lower dimensional simulation in Figure 4. We can see that parameterization once again performs poorly, though, here too the error keeps decreasing. Note that parameterization here can be considered projections of $MP = 10 \times 16 = 160$ dimensional points. Figure 1 too is equivalent to projections of $MP = 40 \times 4 = 160$ dimensional points. The marked difference in the performance suggests that choosing the right projections plays an important role in determining the effectiveness of this technique. Kocis and Whiten [6] have indicated certain poor two-dimensional projections and certain good ones, in a different context, and further study is required into this.

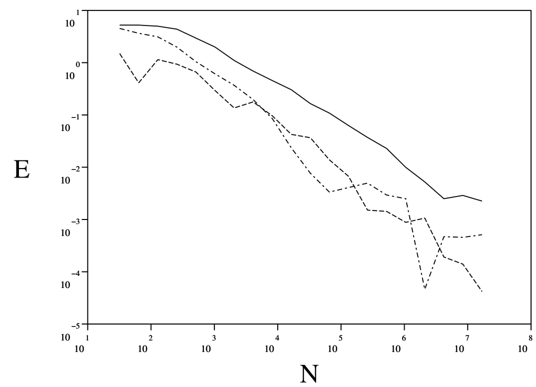

Figure 3. Plot of absolute error (E) vs number of simulations (N) for the Arithmetic Asian option (time steps = 40), using scrambled Halton on 16 processors – equal speeds, parallelized with: leap-frog (dash-dotted line), blocking (dashed) – block size = 10^8, and parameterization (solid).

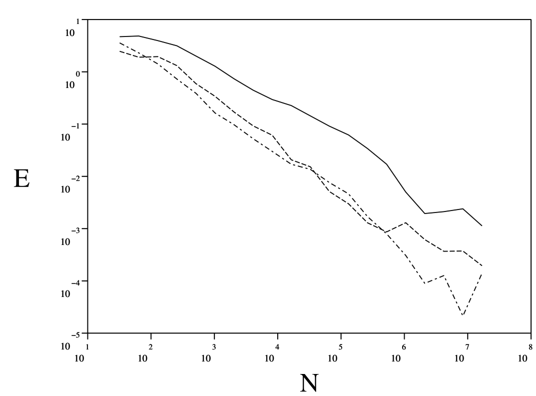

Figure 4. Plot of absolute error (E) vs number of simulations (N) for the Arithmetic Asian option (time steps = 10), using scrambled Halton on 16 processors – equal speeds, parallelized with: leap-frog (dash-dotted line), blocking (dashed) – block size = 10^8, and parameterization (solid).

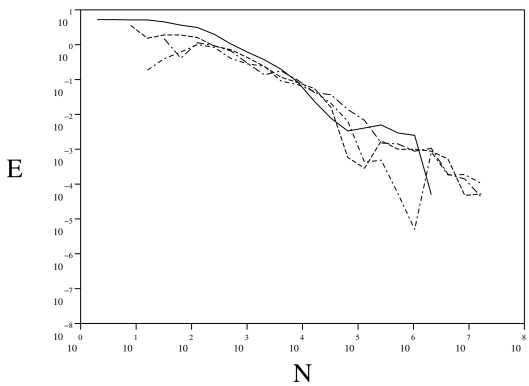

Figure 5. Plot of absolute error (E) vs number of simulations (N) for the Arithmetic Asian option (time steps = 40), using scrambled Halton, parallelized by blocking – block size = 10^8. Number of processors (all equal speeds): 1 (solid line), 4 (dashed), 8 (short-dash dot), and 16 (long-dash dot).

Finally, Figure 5 gives the performance of the blocking scheme for different numbers of processors. We can see that this scheme maintains its performance over the range of P studied. An important disadvantage of the blocking scheme is that if portions of sequences overlap, then the performance can deteriorate, as shown in [9]. Overlap can typically happen if the simulation continues longer than expected, or if the user check-points the state of the computation, and then continues it later, without checking for the block size being exceeded. Yet another limitation of this method is that theoretical asymptotic results are not that useful. While [11] shows that the discrepancy of each block is similar to that of a single LDS, asymptotically, as $N \to \infty$, all sequences overlap, leading to ineffectiveness of parallelization.

5. Conclusions

In summary, we observe that if the simulation is run on a small number of processors with equal speeds, then all the three techniques are about equally effective. If we use a small number of processors with different speeds, then blocking and parameterization are effective, while leap frog gives erroneous results. If we use a large number of processors with equal speeds, then leap frog and blocking are about equally effective, while parameterization converges slower. Leap frog, however, has the advantage that asymptotically, it can be shown to be good in the special case of equal speed, since it will be essentially equivalent to a sequential simulation. If there are a large number of processors running at unequal speeds, then blocking appears the most promising; however, our empirical tests have not tested more than 16 processors.

Defects in the techniques mentioned above may be overcome. For example, one could use leaps other than P with the leap frog scheme, and one could try to use better projections to improve the performance of parameterization. Parameterization has also been shown to be applicable and effective in infinite dimensional simulations in [9]. The blocking scheme appears promising, but we have mentioned possible limitations due to overlap.

References

[1] B. Bromley. Quasirandom number generators for parallel Monte Carlo algorithms. *Journal of parallel and distributed computing*, 38:101–104, 1996.

[2] L. Clewlow and C. Strickland. *Implementing Derivatives Models*. John Wiley & Sons, New York, 1998.

[3] H. Faure. Good permutations for extreme discrepancy. *Journal of Number Theory*, 42:47–56, 1992.

[4] J. C. Hull. *Options, futures, and other derivatives, 3 rd edition*. Prentice-Hall, New Jersey, 2000.

[5] C. Joy, P. P. Boyle, and K. S. Tan. Quasi-Monte Carlo methods in numerical finance. *Management Science*, 42(6):926–938, 1996.

[6] L. Kocis and W. Whiten. Computational investigations of low-discrepancy sequences. *ACM transactions on Mathematical software*, 23(2):266–294, 1997.

[7] J. X. Li and G. L. Mullen. Parallel computing of a quasi-Monte Carlo algorithm for valuing derivatives. *Parallel Computing*, 26:641–653, 2000.

[8] M. Mascagni and A. Srinivasan. `SPRNG`: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[9] G. Ökten and A. Srinivasan. Parallel quasi-Monte Carlo methods on a heterogeneous cluster. In H. N. et al., editor, *Monte Carlo and Quasi-Monte Carlo Methods 2000 (to appear)*. Springer, 2001.

[10] W. Schmid and A. Uhl. Parallel Quasi-Monte Carlo integration using (t,s)-sequences. *Lecture notes in Computer Science*, 1557:96–106, 1999.

[11] W. Schmid and A. Uhl. Techniques of parallel Quasi-Monte Carlo integration with digital sequences and associated problems. *Mathematics and computers in simulation*, 55:249–257, 2000.

[12] A. Srinivasan, D. Ceperley, and M. Mascagni. Random number generators for parallel applications. In *Monte Carlo Methods in Chemical Physics*, volume 105 of *Advances in Chemical Physics*, pages 13–36. John Wiley and Sons, New York, 1999.

[13] R. K. Thulasiram, L. Litov, H. Nojumi, C. T. Downing, and G. R. Gao. Multithreaded algorithms for pricing a class of complex options. In *Proceedings of the 15 th International Parallel and Distributed Processing Symposium (IPDPS01)*. IEEE, 2001.