A Parallel Quasi-Monte Carlo Approach to Pricing Multidimensional American Options

Justin W.L. Wan*
School of Computer Science,
University of Waterloo, Canada
E-mail: jwlwan@uwaterloo.ca
*Corresponding author

Kevin Lai
School of Computer Science,
University of Waterloo, Canada
E-mail: kc3lai@uwaterloo.ca

Adam W. Kolkiewicz
Department of Statistics and Actuarial Science,
University of Waterloo, Canada
E-mail: wakolkiewicz@uwaterloo.ca

Ken Seng Tan
Department of Statistics and Actuarial Science,
University of Waterloo, Canada
E-mail: kstan@uwaterloo.ca

Abstract: In this paper, we develop parallel algorithms for pricing American options on multiple assets. Our parallel methods are based on the low discrepancy (LD) mesh method which combines the quasi-Monte Carlo technique with the stochastic mesh method. We present two approaches to parallelize the backward recursion step, which is the most computational intensive part of the LD mesh method. We perform parallel run time analysis of the proposed methods and prove that both parallel approaches are scalable. The algorithms are implemented using MPI. The parallel efficiency of the methods are demonstrated by pricing several American options, and near optimal speedup results are presented.

Keywords: Quasi-Monto Carlo methods; American options; parallel computing.

Reference to this paper should be made as follows: Wan, J.W.L., Lai, K., Kolkiewicz, A.W. and Tan, K.S. (2006) 'A Parallel Quasi-Monte Carlo Approach to Pricing Multi-dimensional American Options', Int. J. High Performance Computing and Networking, Vol. X, Nos. 1/2/3, pp.XX-XX.

Biographical notes: Justin W.L. Wan is an Assistant Professor at the School of Computer Science from University of Waterloo, Waterloo, Canada. He obtained his PhD degree in Applied Mathematics from University of California, Los Angeles, USA, in 1998. His research interests are in parallel computing, numerical algorithms, and computational finance.

1 Introduction

Monte Carlo simulation is an important computational tool in modern risk management since it is well suited for dealing with the large number of variables that are required to analyze the market risk and credit risk of large portfolios. In fact, the method is really the only viable numerical technique for high dimensional problems and such problems are becoming more prevalent in modern finance.

Despite significant progress, however, the valuation of American options by simulation still remains a challenging problem. One difficulty in applying the Monte Carlo method stems from its early exercise feature. At each point the holder of an American option has to decide whether to exercise the contract or continue to hold on to it. A rational investor will select an optimal exercise strategy that will maximize the value of the contract.

A variety of approaches have been used to value the early exercise feature by simulation. Typically they involve some technique for approximating the early exercise boundary or approximating the transitional density function. This problem can also be set up in a dynamic programming framework where we can solve the optimization problem by working backwards through time. An important advantage of this method over other methods is that it does not require any initial knowledge of the shape of the exercise region, which in a multi-asset problem can be very complex.

An important contributions to this problem has been made by Broadie and Glasserman (1997a), who proved that for a large class of problems the simulation estimator must be biased. They also developed a simulated tree approach which produces two estimators: one biased high and the other biased low. Unfortunately, this algorithm becomes computational burdensome when the number of exercise points or underlying assets is large. Subsequently, they propose a stochastic mesh method (Broadie and Glasserman, 1997b) which could handle much higher dimensions. The rate of convergence for this method, however, is very slow and significant variance reduction techniques need to be incorporated in order for this method to be practically useful.

As inspired by the stochastic mesh method, Boyle et al. (2000) showed that with an appropriate choice of mesh density, the method can be combined with the quasi-Monte Carlo technique to achieve a significant bias reduction of the high-biased estimator. In subsequent papers (Boyle et al. (2002, 2003)), this approach has been extended to low-biased estimators.

Results of an extensive simulation study presented by Boyle et al. (2000, 2002, 2003) indicate that an application of low discrepancy sequences can improve dramatically the accuracy of the mesh method: The achieved rate of convergence exceeded significantly the rate that corresponds to pseudo-random numbers. Despite this, however, for options with several assets the required time is still significant. For instance, in Section 3.5, we present an example with 5 assets and 10 exercise times, the sequential run time is more than 22 hours. In more complex cases, it can take even days. However, in practice, people in financial institutions would want real time quotes.

The use of parallel computing to achieve practical run times has been gaining popularity in the area of computational finance. For instance, Gerbessiotis (2004) considers pricing American option on one asset (whereas the present paper considers multi-assets). His parallel algorithm is based on the binomial lattice model. With only one single asset, the complexity is linear in, n, the number of the discretization time step. Hence, the parallel efficiency can only achieve around 50% for the values of n tested in Gerbessiotis (2004). Thulasiram and Bondarenko (2003), on the other hand, consider a more challenging problem in parallelizing the multidimensional binomial lattice model for pricing more complex derivatives. Improved parallel efficiency is reported with increasing number of assets. Both of these papers use MPI (message passing interface) programming model. Multithreaded parallel implementation has also been studied by Thulasiram and Thulasiraman (2003) and Thulasiram et al. (2001).

In this paper, we study parallel implementations of the low discrepancy (LD) mesh method, and in particular, the backward recursion step which is the most computational intensive part of the LD mesh method. We propose two parallel approaches based on the block and checkerboard layout of the data. The idea of the block approach is to partition data among processors arranged in a linear array topology whereas the checkerboard approach partitions data in a 2D array. Thus the communication cost is relatively cheaper for the latter. Furthermore, since the complexity of the backward recursion is quadratic in b, the number of asset prices per exercise date, it is then more amenable to parallelization and more effective use of multiple processors can be resulted. The numerical examples conducted in 3.5 indicate that our implementation can achieve almost optimal speedup.

The paper is organized as follows. In Section 2, we describe the low discrepancy mesh method. In Section 3, we present the the proposed block and checkerboard parallel implementations of the LD mesh method. Section 3.5 shows the numerical results and parallel efficiency. Section 5 concludes the paper.

2 Low Discrepancy Mesh Method

The objective is to price an American style derivative security contingent on n underlying asset prices $\{\vec{S}_t\} = \{(S_t^1, S_t^2, \ldots, S_t^n)\}$. The security can be exercised prior to maturity at $d+1$ time points (including the initial time), which will be denoted as $t_0, t_1, \ldots, t_d = T$. If the security is exercised at time τ, its value is equal to $I(\tau, \vec{S}_\tau)$, where $I(\cdot)$ is a known function representing the discounted value of the contract.

We assume that under the risk-neutral measure Q, the price process $\{\vec{S}_t\}$ is a Markov process with a fixed initial state, $\vec{S}_0$, whose transition probability densities $f(t_i, t_j, \vec{x}; \cdot)$, defined by $P(\vec{S}_{t_j} \in A | \vec{S}_{t_i} = \vec{x}) = \int_A f(t_i, t_j, \vec{x}; \vec{u}) d\vec{u}$, exist and are known.

The valuation of the American derivative can be formulated as the maximization of its expected exercise value taken over all stopping times taking values in the set $\{t_0, t_1, \ldots, T\}$. It is well known that this problem can be solved using the principle of dynamic programming: We find the functions $V(t_i, \cdot)$, the value of the American option, through the backward recursion:

$$\begin{aligned} V(T, \vec{x}) &= I(T, \vec{x}), \\ V(t_i, \vec{x}) &= \max[I(t_i, \vec{x}),\ C(t_i, \vec{x})], \quad i = d-1, ..., 0, \end{aligned}$$

where $C(t_i, \vec{x})$ is the continuation value at point $\vec{x}$

$$C(t_i, \vec{x}) = E[V(t_{i+1}, \vec{S}_{t_{i+1}}) | \vec{S}_{t_i} = \vec{x}]), \tag{1}$$

and finally set $V = V(0, \vec{S}_0)$.

To use this method in practice we must be able to calculate or approximate efficiently the continuations values $C(t_i, \vec{x})$ for all $i = 0, ..., d-1$ and some selected set of points $\vec{x}$ from the state space. For $t = t_0$ this can be accomplished using the Monte Carlo method:

$$E[V(t_1, \vec{S}_{t_1}) | \vec{S}_{t_0}] \simeq \frac{1}{b} \sum_{l=1}^{b} \hat{V}(t_1, \vec{X}_{t_1}(l)),$$

where $\hat{V}(t_1, \cdot)$ is an approximation of $V(t_1, \cdot)$ obtained from the backward recursion and $\vec{X}_{t_1}(1), \ldots, \vec{X}_{t_1}(n)$ is a random sample drawn from a distribution with density function $f(t_0, t_1, \vec{S}_{t_0}; \cdot)$. In practice, however, this method will quickly become unworkable because the number of points will grow exponentially.

The key to the stochastic mesh method (Broadie and Glasserman, 1997b) and low discrepancy (LD) mesh method (Boyle et al. (2000, 2002, 2003)) is the observation that points that we generate to calculate one conditional expectation can also be used to calculate other expectations. In both methods, first we generate a mesh of state points $\{\vec{X}_{t_i}(j), i = 1, \ldots, d, j = 1, \ldots, b\}$, from $\mathcal{R}^s$. In the stochastic mesh method this is achieved by generating random points from a distribution specified by a certain density function g_{t_i}, which is referred to as a mesh density. For the low discrepancy mesh method the mesh points are generated by applying the inverse method to a low discrepancy sequence. Once a mesh is constructed, the values $\hat{V}(t_i, \cdot)$ are calculated using the backward recursion method, where the continuation values are approximated by

$$\hat{C}(t_i, \vec{X}_{t_i}(j)) = \frac{1}{b} \sum_{l=1}^{b} \hat{V}(t_{i+1}, \vec{X}_{t_{i+1}}(l)) w(t_i, \vec{X}_{t_i}(j), \vec{X}_{t_{i+1}}(l)), \tag{2}$$

for $j = 1, \ldots, b$, $i = 0, 1, \ldots, d$, where $w(t_i, \vec{x}, \vec{y}) := [f(t_i, t_{i+1}, \vec{x}; \vec{y})]/g_{t_i}(\vec{y})$ is the Radon-Nikodym derivative. The introduction of the weights w is necessary as the points in the mesh are sampled from the density g_{t_i}. This procedure, applicable to both stochastic and deterministic mesh points, generates an estimate that is biased high.

A good choice of the densities g_{t_i}, $i = 1, \ldots, d$, is crucial for the success of the mesh method. For random sampling, Broadie and Glasserman (1997b) suggests to use a uniform mixture of transition densities. More formal argument for American options in favor of this distribution is presented by Boyle et al. (2000). However, in the same paper it has been also shown that for low discrepancy sequences a simpler choice of the mesh density g equal to the marginal distribution $f(0, t_i, S_0; \cdot)$ will give a very similar rate of convergence.

3 Parallel implementation

In this section, we introduce two parallel implementations of the LD mesh method. The first is an intuitive block approach, and the other one is the more communication efficient checkerboard approach.

The general LD mesh method can be summarized into three steps as follows (note: we simplify the notation from t_i to t here for easy exposition).

- Step 1: mesh generation
  - Start with $X_0(1) = S_0$.
  - Generate $X_t(i) = (X_t^1(i), X_t^2(i), \ldots, X_t^n(i))$, $i = 1, 2, \ldots, b$; $t = 1, 2, \ldots, T$.
- Step 2: initialization of estimators at final time T
  - $\hat{V}(T, X_T(i)) = I(T, X_T(i))$, $i = 1, 2, \ldots, b$.
- Step 3: backward recursion
  - $\hat{V}(t, X_t(i)) = \max(I(t, X_t(i)), \hat{C}(t, X_t(i)))$, $i = 1, 2, \ldots, b$; $t = T-1, \ldots, 0$.
  - $\hat{C}(t, X_t(i)) = \frac{1}{b} \sum_{j=1}^{b} \hat{V}(t+1, X_{t+1}(j)) w(t, i, j)$.
  - $w(t, i, j)$=weight attached to the arc joining $X_t(i)$ to $X_{t+1}(j)$.

The complexity of Step 1 and Step 2 are $O(nbd)$ and $O(b)$, respectively, whereas the complexity for Step 3 is $O(nb^2d)$. Thus, the backward recursion step, in particular, the computation of $\hat{C}$, contributes the most computational time. This is also consistent with our findings empirically that over 90% of the computational time is spent in the backward recursive step. Furthermore when we increase both n and d, b will need to be increased accordingly in order to achieve a reasonable accuracy. Consequently, the overall computational time grows much worse than linearly. Hence this put additional computational burden on the underlying LD mesh method. For this reason, the primary focus of this paper is to address the parallelization issues of the backward recursion.

3.1 Mesh generation

At the beginning, the total work of generating the entire mesh is divided up among all the processors. Each processor is responsible for generating the asset prices for a certain period of time, then send the simulated asset prices to other processors.

Let $\vec{x}_j, j = 1, \ldots, bd$, be a sequence of low discrepancy points used to simulate the mesh points. To parallelize

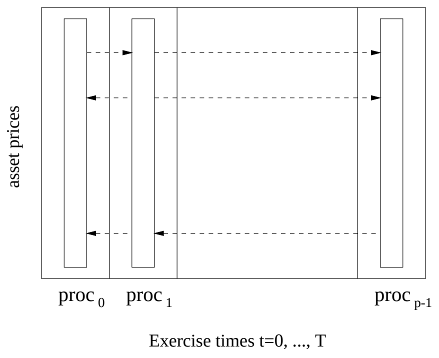

Figure 1: Data partition of all asset prices at all times among p processors. The arrows show the data flow of the all-to-all broadcast.

this procedure, each processor is assigned with a block of elements from the low discrepancy sequence. For example, a processor, $proc_k$, has $x_{kB+1}, x_{kB+2}, x_{kB+3}, \cdots$, where $B = bd/p$ is the block size; see e.g. Bratley and Fox (1988); Bromley (1996); Li and Mullen (2000); Okten and Srinivasan (2002); Srinivasan (2002). Each processor generates a block of asset prices for a period of time. When finished, perform an all-to-all broadcast so that each processor has a complete mesh (of all the asset prices at all times) for the backward recursion; see Fig. 1. The memory storage is nbd.

We remark that in the LD mesh algorithm, we only need to generate the mesh once. This mesh generation step typically is negligible compared to the backward recursion.

3.2 Backward recursion: block approach

As argued above, this step dominates the overall computational time. At each exercise time step t, the processors need to determine $\hat{C}(t, X_t(i))$ (and hence $\hat{V}(t, X_t(i))$), $i = 1, 2, \ldots, b$, using (2). To parallelize this calculation, the idea underlies the block approach is to distribute the computation of $\hat{C}(t, X_t(i))$ to each processor. More precisely, let $\{I_k\}$ be a partition of $\{1, 2, \ldots, b\}$; i.e. $\{1, 2, \ldots, b\} = I_0 \cup I_1 \cup \cdots \cup I_{p-1}$, where $I_k = \{kb/p + 1, \ldots, (k+1)b/p\}$ and p=total number of processors. Then, each processor, $proc_k$, will compute $\hat{C}(t, X_t(i)), i \in I_k$ at the mesh point in the range specified by I_k in parallel.

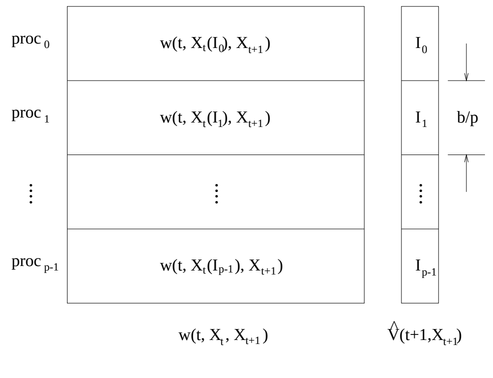

Figure 2: Data partition of w and $\hat{V}$ among p processors.

The computation of $\hat{C}(t, X_t(i))$ consists of 3 steps.

Step 1: initialization of w and $\hat{V}$

The weight function w and the approximate option function $\hat{V}$ computed from the previous time step are partitioned among processors as shown in Fig. 2. Note that w depends on the current asset prices X_t as well as previous asset prices X_{t+1}, and so its values are stored in a 2D array. In this partition, $proc_k$ contains those weights w and $\hat{V}$ whose current asset prices are in the range $X_t(j)$, $j \in I_k$. Also, each processor has the entire mesh with all the asset prices at all times, and so each processor can compute their own w values without any communication.

Step 2: distribution of $\hat{V}(t+1, X_{t+1}(\cdot))$ among processors

We note that we need $\hat{V}(t+1, \cdot)$ at *all* asset prices to compute $\hat{C}(t, X_t(i))$. Thus, we perform an all-to-all broadcast as shown in Fig. 3. As a result, every processor now has a copy of entire $\hat{V}(t+1, \cdot)$.

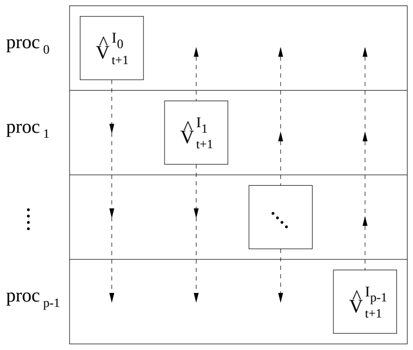

Figure 3: Distribution of $\hat{V}(t+1, X_{t+1}(\cdot))$ among all processors. The symbol $\hat{V}_{t+1}^{I_k}$ denotes the values $\hat{V}(t+1, X_{t+1}(i))$, $i \in I_k$. The arrows show the data flow of the all-to-all broadcast.

Step 3: local computation

Finally, each processor $proc_k$ computes

$$\hat{C}(t, X_t(i)) = \frac{1}{b} \sum_{j=1}^{b} \hat{Q}(t+1, X_{t+1}(j)) w(t, i, j)$$

using the local data it has without communication. Once $\hat{C}(t, X_t(i))$, $i \in I_k$, are known, we can compute $\hat{V}(t, X_t(i))$ accordingly.

Note: In Step 2, the all-to-all broadcast of $\hat{V}(t+1, X_{t+1}(\cdot))$ can also be done at the end of Step 3 whenever the current $\hat{V}(t, X_t(\cdot))$ are available.

The block parallel backward recursion algorithm can be summarized as:

```
Algorithm: (at each exercise time $t$ until $t = 0$)

for each $proc_k$, $i \in I_k$:
    compute $w(t, X_t(i), X_{t+1}(j)) = \frac{f(t,X_t(i),X_{t+1}(j))}{g(t+1,X_{t+1}(j))}$, $j = 1, 2, \ldots, b$
    compute $\hat{C}(t, X_t(i)) = \frac{1}{b}\sum_{j=1}^{b} \hat{V}(t+1, X_{t+1}(j)) w(t, X_t(i), X_{t+1}(j))$
    compute $\hat{V}(t, X_t(i)) = \max(I(t, X_t(i)), \hat{C}(t, X_t(i))$
    all-to-all broadcast local $\hat{V}(t, X_t(i))$ to other procs
```

Remark: Since each processor has all the asset prices at all times, w can be computed without any communication. Hence, communication only occurs at the end of each exercise time.

3.3 Backward recursion: checkerboard approach

In the parallel block approach, the communication is all concentrated in the all-to-all broadcast. Although it only occurs once every time step, all-to-all broadcast is relatively complex and time-consuming mode of communication. In the parallel performance analysis (Section 3.5), the large communication time cost shows up in the overall parallel run time. For parallel platforms with many processors and high latency, e.g. PC clusters, it is desirable to avoid all-to-all broadcasts.

In this section, we introduce the checkerboard parallel implementation, namely, partition data into a checkerboard fashion. Using this special layout, we are able to reduce the communication costs between processors and ultimately reduce the overall computational time. In this paper, we assume that the number of processors is a square number for easy implementation. In practice, however, we can relax on this condition.

The basic idea is to arrange the processors in a two-dimensional mesh (see Fig. 4), and that each processor only has a portion of $w(t, X_t(\cdot), X_{t+1}(\cdot))$. Recall that the computation of $\hat{C}(t, X_t(i))$ involves a sum of weighted $\hat{Q}(t+1, X_{t+1}(j))$. In contrast with the block approach, each processor computes only a local partial sum of $\hat{C}(t, X_t(i))$ and then at the end, the partial sums across each row of processors are combined to obtain $\hat{C}(t, X_t(i))$.

Specifically, we introduce another partition of $\{1, 2, \ldots, b\}$; namely $\{1, 2, \ldots, b\} = J_0 \cup J_1 \cup \cdots \cup J_{\sqrt{p}-1}$, where $J_k = \{kb/\sqrt{p}, \ldots, (k+1)b/\sqrt{p}\}$. Then we break down the sum for computing $\hat{C}(t, X_t(i))$ into $\sqrt{p}$ partial sums:

$$\hat{C}(t, X_t(i)) = \tag{3}$$
$$\frac{1}{b}\left(\sum_{j\in J_0} \hat{V}(t+1, X_{t+1}(j)) w(t, X_t(i), X_{t+1}(j)) + \cdots \right.$$
$$\left.\sum_{j\in J_{\sqrt{p}-1}} \hat{V}(t+1, X_{t+1}(j)) w(t, X_t(i), X_{t+1}(j))\right).$$

We remark from (3) that processors in the kth column of the checkerboard data partition need the values of $\hat{V}(t+1, X_{t+1}(j))$, $j \in J_k$, in order to compute the kth partial sum of $\hat{C}(t, X_t(i))$. Also, processors in the lth row are responsible for computing $\hat{C}(t, X_t(i))$ at the asset prices specified by $i \in J_l$.

Before the start of the backward recursion process, the diagonal processors compute $\hat{V}(T, X_T(j))$, $j \in J_k$, if that processor is located at column k. During the backward recursion steps, the computation of $\hat{C}(t, X_t(i))$ consists of the following 4 steps.

Step 1: initial partition of w and $\hat{V}$
The 2D array of the weight function values $w(t, X_t, X_{t+1})$ are partitioned among p processors organized as a 2D mesh, or checkerboard, as shown in Fig. 4. Thus, if a processor is located in the lth row and kth column of the checkerboard partition, then it has $w(t, X_t(i), X_{t+1}(j))$, $i \in J_l$, $j \in J_k$. The values of $\hat{V}(t+1, X_{t+1})$ are distributed only among processors on the diagonal; i.e. the diagonal processor on row k will have $\hat{V}(t+1, X_{t+1}(i))$, $i \in J_k$.

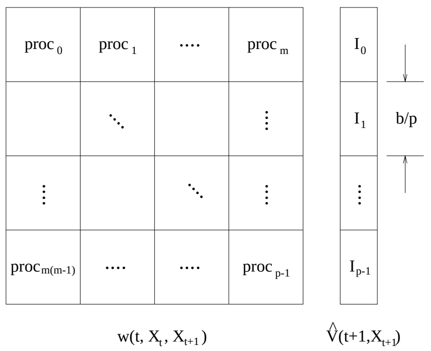

Figure 4: Data partition of w and $\hat{V}$ among p processors. Here, $m = \sqrt{p}$.

Step 2: distribution of $\hat{V}(t+1, X_{t+1})$ along each column
As pointed out above, processors in the kth column need the values $\hat{V}(t+1, X_{t+1}(j))$, $j \in J_k$, which are store in the diagonal processor. Thus we perform an one-to-all broadcast from the diagonal processor to the processors located in the same column; see Fig. 5.
Step 3: local computation
Each processor in row k and column l computes the local partial sum in parallel:

$$\hat{C}_i^{J_l} = \frac{1}{b}\sum_{j\in J_l} \hat{V}(t+1, X_{t+1}(j)) w(t, X_t(i), X_{t+1}(j)), i \in J_k.$$

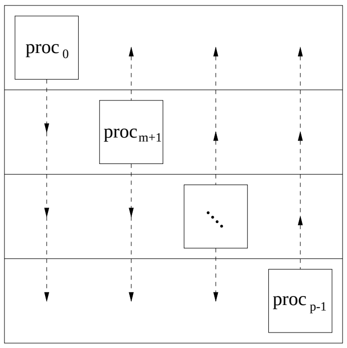

Figure 5: Distribution of $\hat{V}(t+1, X_{t+1}(\cdot))$ among all processors. Here $m = \sqrt{p}$. The arrows show the data flow of the one-to-all broadcasts.

Step 4: accumulation of partial sums
Finally, we compute $\hat{C}(t, X_t(i))$, $i \in J_k$, by combining the partial sums across processors in row k:

$$\hat{C}(t, X_t(i)) = \sum_{l=0}^{\sqrt{p}-1} \hat{C}_i^{J_l}.$$

To do so, all-to-one reductions are performed from processors across the rows to the diagonal processors as shown in Fig. 6. At the end, the values $\hat{V}(t, X_t(i))$ are also computed. Since only the diagonal processors have $\hat{C}(t, X_t(i))$ and hence the new $\hat{V}(t, X_t)$, and we are now back to Step 1 for the next exercise time.

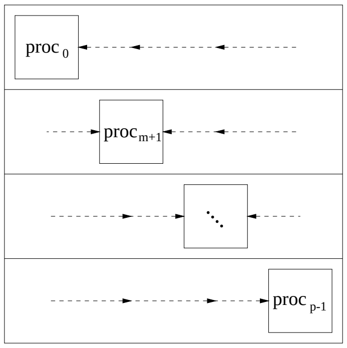

Figure 6: Accumulation of partial sums to compute $\hat{C}(t, X_t(i))$. Here $m = \sqrt{p}$. The arrows show the data flow of the all-to-one reductions.

3.4 Memory optimization

The above implementations (block or checkerboard) use nbd memory space to store the complete mesh (i.e. all asset prices at all times). The memory consumption might not be an issue if n, b, d are small; however, due to the ability that the parallel implementations can solve large size problems, it may not be desirable to store the entire mesh in memory.

This can be optimized since the asset prices in the mesh are independent of one another and therefore at time t, only mesh points of t, $t+1$ are needed. Thus, we only generate new mesh points in each exercise point and in the mean time discard the old mesh points in previous exercise times. As a result, our *mesh-on-demand* optimization only stores $2nb$ mesh points (vs nbd mesh points in a complete mesh).

3.5 Parallel performance analysis

In pricing American options with many assets and number of exercise dates, large number of processors would be necessary to achieve fast run times. However, the associated communication cost in the parallel algorithm will increase as well, which deteriorates the overall efficiency. Thus, it is desirable the parallel algorithms are scalable (Grama et al., 2003), in the sense that the parallel efficiency can be kept constant by simultaneously increase the number of processors and the size of the problem.

To measure the scalability of the block and checkerboard approaches, we analyze the parallel run times of these two algorithms. For easy exposition, we adopt a simplified time model for communication (Grama et al., 2003). More precisely, the communication time for sending a message of length m between two processors is given by:

$$T_{comm} = T_s + mT_w,$$

where T_s is the startup time (or latency), and T_w is the transfer time per word.

For the block approach, an all-to-all broadcast is performed in each exercise time. Thus the parallel run time is:

$$T_p^{block} = \frac{b^2}{p} + \log p\, T_s + \frac{b(p-1)}{p}\, T_w,$$

and so the parallel efficiency of the algorithm is then given by:

$$E_p^{block} = \frac{b^2}{b^2 + p\log p\, T_s + b(p-1)\, T_w}.$$

On the other hand, for the checkerboard approach, only an one-to-all and all-to-one broadcasts are performed. Thus the parallel run time is reduced to:

$$T_p^{chkrbrd} = \frac{b^2}{p} + \log p\, T_s + \frac{b\log p}{\sqrt{p}}\, T_w,$$

and the corresponding parallel efficiency is given by:

$$E_p^{chkrbrd} = \frac{b^2}{b^2 + p\log p T_s + b\sqrt{p}\log p T_w}.$$

By comparing the parallel run times of the block and checkerboard approaches, we see that the checkerboard is more efficient for having a smaller T_w term. Specifically, it gains by a factor of $\sqrt{p}/\log p$. For large p, this factor can be significant as $\log p$ is a very mildly growing function. Moreover, the scalability of the checkerboard approach is also better in the sense that to maintain constant efficiency, or in other words, an effective use of the processors, the block approach requires $b = O(p)$ whereas the checkerboard approach only requires $b = O(\sqrt{p}\log p)$. Thus, the latter impose a less restrictive constraint on the size of the problem to maintain constant efficiency.

4 Numerical results

In this section, we demonstrate the accuracy and efficiency of our parallel LD mesh methods. The parallel algorithms are implemented using C and MPI. Thus, our code is portable on a wide range of parallel platforms. The programs are run on a SGI Onyx shared memory machine with 8 processors. (We note that OpenMP is an alternative implementation of the parallel algorithms presented on the SGI machine; however, we opted to use MPI for portability reason.) Timing is obtained via the MPI wall clock function, *MPI_Wtime()*.

We consider the following American options:

1. American call option on one asset with 50 exercise dates and payoff = $\max(S-K,0)$. The parameter values are $d = 50$, $r = 5\%$, $\delta = 10\%$, $T = 3$ years, $K = 100$, $S_0 = \{90, 100, 110\}$, $\sigma = \{20\%, 40\%\}$, $b = \{256, 1024, 4096\}$.

2. American maximum call option on five assets with 4 exercise dates and payoff = $\max(\max(S_t^1, \ldots, S_t^5) - K, 0)$. The parameter values are $d = 3$, $r = 5\%$, $\delta = 10\%$, $\sigma = 20\%$, $T = 3$ years, $K = 100$, $S_0 = 100$, $b = \{4096, 8192, 16384\}$, and the assets are uncorrelated.

3. Similar to the last example except with 10 exercise dates (i.e. $d = 9$).

4. American geometric average call option on five assets with 11 exercise dates and payoff = $\max((S_t^1 \cdots S_t^5)^{1/5} - K, 0)$. The parameter values are $d = 10$, $r = 3\%$, $\delta = 5\%$, $\sigma = 40\%$, $T = 1$ year, $K = 100$, $S_0 = 100$, $b = \{4096, 8192, 16384\}$, and the assets are uncorrelated.

Among these test cases, the first example is the simplest. Efficient numerical methods such as the binomial model (Cox et al., 1979) exist for valuing such option contract. We included this standard American option in our analysis so that we can use this as a benchmark against the parallel LD mesh method. The second example is a situation where it is computational infeasible to use binomial model. The computational complexity of the binomial model grows exponentially with the number of underlying assets. Researching for efficient numerical methods for pricing multi-asset American options remains an on-going challenging problem. Hence this is an area where the parallel LD mesh method has a potential application. The third and fourth examples are even more challenging in that they have more exercise dates. The last three examples are studied by Broadie and Glasserman (1997a). Because of the significant bias in the underlying stochastic mesh methods, Boyle et al. (2000)) proposed a number of control variates in conjunction of the stochastic mesh method. The parallel LD mesh method, on the other hand, is able to reduce substantially the bias without using any control variate, as illustrated in Table 1.

Table 1: Various mesh estimators for American call option on one asset with 50 exercise dates.

			Mesh estimators		
S_0	σ	b	BG	AH	Parallel LD
90	20%	256	9.96	7.73	4.66
		1024	6.22	5.33	4.51
		4096	5.23	4.94	4.47
		Lattice	4.47	4.47	4.47
90	40%	256	33.07	25.77	15.15
		1024	20.75	17.74	14.55
		4096	17.43	16.31	14.41
		Lattice	14.40	14.40	14.40
100	20%	256	16.30	13.13	8.26
		1024	10.40	9.39	8.16
		4096	9.09	8.74	8.14
		Lattice	8.14	8.14	8.14
100	40%	256	42.38	33.57	19.92
		1024	26.74	23.55	19.36
		4096	22.78	21.53	19.24
		Lattice	19.23	19.23	19.23
110	20%	256	24.14	20.11	13.47
		1024	16.10	14.96	13.43
		4096	14.35	14.00	13.42
		Lattice	13.42	13.42	13.42
110	40%	256	52.44	42.18	25.34
		1024	33.32	29.89	24.84
		4096	28.68	27.27	24.75
		Lattice	24.74	24.74	24.74

Table 1 compares our results (Parallel LD) with the stochastic method method (BG) by Broadie and Glasserman (1997a), and the average mesh estimator (AH) by Avramidis and Hyden (1999). The results from the binomial lattice method (Lattice) are used as benchmark. Each number is obtained by the mean of 10 independent runs and the number in brackets is the sample standard error. The BG mesh estimators are high-biased and the bias is in fact quite high for low mesh points or high volatility cases. The AH mesh estimators show a reduced bias of the BG results, but are still subject to the same high variation as the BG mesh estimators. The convergence of our parallel LD mesh estimators is very fast; accurate answers can generally be obtained by using small number of mesh points. Moreover, the parallel LD mesh method is very stable as indicated by the small standard errors.

We now examine the parallel efficiency of the parallel LD mesh methods by considering American options on 5 assets. For this example, the option estimates (based on 10 independent replications) are: 25.35(0.004), 25.34(0.004), 25.29(0.003) for $b = 4096, 8192, 16384$, respectively. These values are in good agreement with (Broadie and Glasser-

Table 2: The parallel run times (in seconds) of the block approach for pricing American max-option on 5 assets, 4 exercise times.

b	Number of processors			
	1	2	3	4
4096	961.52	484.948	325.195	224.263
8192	3834.285	1917.969	1286.414	962.679
16384	15311.54	7656.253	5129.152	3858.866
b	Number of processors			
	5	6	7	8
4096	198.578	166.481	141.99	125.649
8192	776.43	647.71	557.999	490.994
16384	3090.062	2574.452	2212.128	1942.534

Table 3: The parallel run times (in seconds) of the checkerboard approach for pricing American max-option on 5 assets, 4 exercise times.

b	Number of processors	
	1	4
4096	1315.259	324.632
8192	5115.727	1285.494
16384	20453.7	5118.776

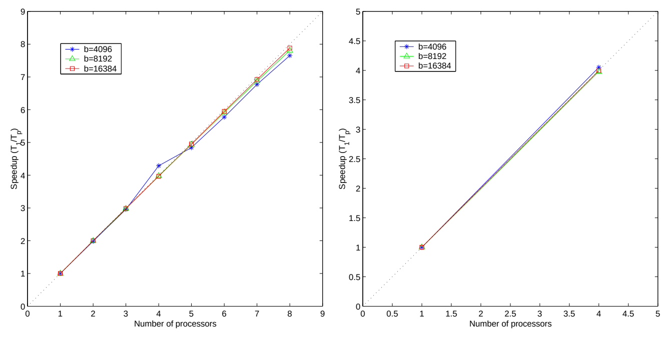

Figure 7: Speedup graph of the block approach (left) and the checkerboard approach (right), for pricing American max-option on 5 assets, 4 exercise times.

man, 1997b, Table 4). It should be emphasized that the reported results in Broadie and Glasserman (1997b) are based on a number of control variates while our results is a direct implementation of LD mesh method. Additional enhancement to the underlying LD mesh method can similarly be achieved with the variance reduction techniques such as incorporating the control variates (see Boyle et al. (2002)).

We compute the option pricing using different number of processes, $p = 1, 2, \ldots, 8$. The timing results (measured in seconds) are shown in Table 2 and Table 3 for the block and checkerboard approaches, respectively. For the checkerboard approach, since the processors are mapped onto a square mesh topology, we can test it only using 1 and 4 processors.

To analyze their parallel efficiency, we use the parallel timings to produce speedup plots for the two parallel implementations. As shown in Fig. 7, both block and checkerboard approaches achieve almost the theoretical optimal speedup (indicated by the dotted line). We remark that the performance gain of the checkerboard approach is not dramatic as the communication costs of all-to-all and all-to-one (or one-to-all) broadcasts are not very different among 8 processors where latency is low (shared-memory architecture). However, if more processors were to be used on a distributed system where latency is much higher, we would expect to have more significant differences.

Table 4: The parallel run times (in seconds) of the block approach for pricing American max-option on 5 assets, 10 exercise times.

b	Number of processors			
	1	2	3	4
4096	5095.13	2545.81	1706.74	1286.18
8192	20377.22	10181.06	6826.20	5153.52
16384	81533.66	40674	27270.57	20611.08
b	Number of processors			
	5	6	7	8
4096	1059.81	870.29	745.50	656.66
8192	4233.41	3487.77	2975.32	2626.94
16384	16881.57	13891.15	11831.57	10454.64

Table 5: The parallel run times (in seconds) of the checkerboard approach for pricing American max-option on 5 assets, 10 exercise times.

b	Number of processors	
	1	4
4096	5121.534	1290.33
8192	20501.65	5159.38
16384	81862.02	20623.53

The third option is similar to the second option except that we increase the complexity by increasing the num-

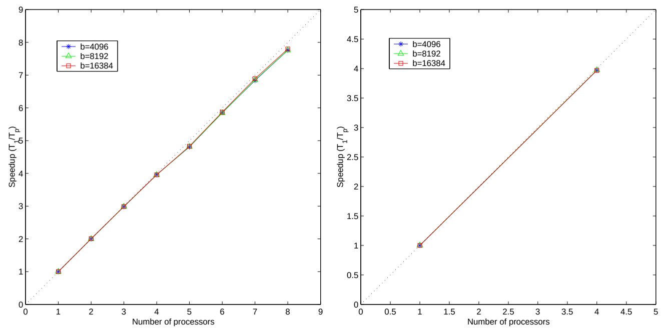

Figure 8: Speedup graph of the block approach (left) and the checkerboard approach (right), for pricing American max-option on 5 assets, 10 exercise times.

ber of exercise dates to 10. In this case, the option estimates are: 27.59(0.169), 27.11(0.047), 26.72(0.022) for $b = 4096, 8192, 16384$, respectively. The fourth example is American geometric average call option, and the number of exercise dates is increased to 11. The option estimates computed are: 4.48(0.024), 4.40(0.007), 4.34(0.000) for $b = 4096, 8192, 16384$, respectively. Again, they are consistent with those reported in (Broadie and Glasserman, 1997b, Table 6, 7).

The parallel timing results for the third option are shown in Table 4 and Table 5, and the corresponding speedup graphs are shown in Fig. 8. The results for the fourth option are shown in Table 6 and 7, and in Fig. 9. We see that as in the previous example, both the block and checkerboard approaches again exhibit near optimal speedup.

Table 6: The parallel run times (in seconds) of the block approach for pricing American geometric average option on 5 assets, 11 exercise times.

b	Number of processors			
	1	2	3	4
4096	5825.38	2920.18	1949.71	1467.83
8192	23290.34	11630.76	7748.60	5836.18
16384	92947.7	46485.45	30957.5	23296.62
b	Number of processors			
	5	6	7	8
4096	1212.89	994.59	869.83	772.20
8192	4804.93	3913.95	3377.87	2951.25
16384	19071.21	15705.02	13366.35	11946.57

5 Conclusion

We have presented a block and checkerboard parallel implementations of the LD mesh method for pricing American options on multi-assets. We have analyzed the parallel run times of these approaches based on a simplified communication cost model and proved that these two methods are scalable; that is, they can maintain constant parallel efficiency if the number of processors and the size of the problem are increased accordingly. We have validated the accuracy of our parallel algorithms for pricing the American call option on one asset. We then demonstrated the parallel efficiency of our parallel algorithms for the case of multiple assets. Near optimal speedup results are obtained on a SGI Onyx machine with 8 processors. In the future, we would like to compare the parallel efficiency of our block and checkerboard approaches on distributed systems where latency is high.

Table 7: The parallel run times (in seconds) of the checkerboard approach for pricing American geometric average option on 5 assets, 11 exercise times.

b	Number of processors	
	1	4
4096	5870.807	1500.41
8192	23402.88	5948.86
16384	93263.29	23693.95

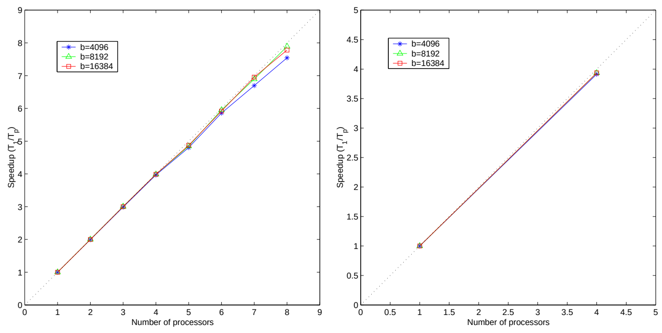

Figure 9: Speedup graph of the block approach (left) and the checkerboard approach (right), for pricing American geometric average option on 5 assets, 11 exercise times.

REFERENCES

Avramidis, A. N. and Hyden, P. (1999). Efficiency improvement for pricing American options with a stochastic mesh. In Garrington, P., Nembhard, H., Sturrock, D., and Evans, G., editors, *Proceedings of the 1999 Winter Conference*, pages 344–350.

Boyle, P. P., Kolkiewicz, A., and Tan, K. S. (2000). Pricing American style options using low discrepancy mesh methods. Technical Report IIPR 00-07, University of Waterloo.

Boyle, P. P., Kolkiewicz, A., and Tan, K. S. (2002). Pricing American derivatives using simulation: a biased low approach. In Fang, K.-T., Hickernell, F., and Niederreiter, H., editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, Berlin. Springer-Verlag.

Boyle, P. P., Kolkiewicz, A., and Tan, K. S. (2003). An improved simulation method for pricing high-dimensional American derivatives. *Mathematics and Computers in Simulation*, 62(3-6):315–322.

Bratley, P. and Fox, B. L. (1988). Algorithm 659 implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software*, 14:88–100.

Broadie, M. and Glasserman, P. (1997a). Pricing American-style securities using simulation. *Journal of Economic Dynamics and Control*, 21(8-9):1323–1352.

Broadie, M. and Glasserman, P. (1997b). A stochastic mesh method for pricing high-dimensional American options. Working paper, Columbia University.

Bromley, B. C. (1996). Quasirandom number generators for parallel Monte Carlo algorithms. *Journal of Parallel and Distributed Computing*, 38:101–104.

Cox, J. C., Ross, S. A., and Rubinstein, M. (1979). Option pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263.

Gerbessiotis, A. V. (2004). Architecture independent parallel binomial tree option price valuations. *Parallel Computing*, 30:303–318.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison Wesley.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, second edition.

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

Okten, G. and Srinivasan, A. (2002). Parallel quasi-Monte Carlo methods on a heterogeneous cluster. In Fang, K.-T., Hickernell, F., and Niederreiter, H., editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, Berlin. Springer-Verlag.

Srinivasan, A. (2002). Parallel and distributed computing issues in pricing financial derivatives through quasi-Monte Carlo. In *Proceedings of the Sixteenth Interanational Parallel and Distributed Processing Symposium*, Fort Lauderdale, USA.

Thulasiram, R. K. and Bondarenko, D. A. (2003). Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. *To appear in Parallel and Distributed Computing Practices.*

Thulasiram, R. K., Litov, L., Nojumi, H., Downing, C. T., and Gao, G. R. (2001). Multithreaded algorithms for pricing a class of complex options. In *IEEE/ACM International Parallel and Distributed Processing Symposium.*

Thulasiram, R. K. and Thulasiraman, P. (2003). Performance evaluation of a multithreaded fast Fourier transform algorithm for derivative pricing. *Journal of Supercomputing*, pages 43–58.