Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 33 (2007) 92–108
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Parallel computing in Asian option pricing

Halis Sak [a,b], Süleyman Özekici [c], İlkay Boduroğlu [d,*]

[a] *Boğaziçi University, Department of Industrial Engineering, Bebek-İstanbul, Turkey*
[b] *İstanbul Kültür University, Department of Industrial Engineering, Ataköy-İstanbul, Turkey*
[c] *Koç University, Department of Industrial Engineering, Sarıyer-İstanbul, Turkey*
[d] *İstanbul Technical University, Informatics Institute, Maslak-İstanbul, Turkey*

Available online 26 December 2006

Abstract

We discuss the use of parallel computing in Asian option pricing and evaluate the efficiency of various algorithms. We only focus on "backward-starting fixed strike" Asian options that are continuously averaged. We implement a partial differential equation (PDE) approach that involves a single state variable to price the Asian option, and implement the same methodology to price a standard European option to check for accuracy. A parabolic PDE is solved by using both explicit and Crank–Nicolson's implicit finite-difference methods. In particular, we look for algorithms designed for implementing the computations in massively parallel processors (MPP). We evaluate the performance of the algorithms by comparing the numerical results with respect to accuracy and wall-clock time of code executions. Codes are executed on a Linux PC cluster.

Keywords: Asian option pricing; Computational finance; Parallel computing; Finite-difference methods

1. Introduction

An option is a derivative security which gives its holder the right to receive a payoff contingent on a primary security within a specified period of time. Exotic options which have more complicated payoffs than standard European and American options are extensively used in finance in order to meet specific needs of customers. The complexity of the payoff structure makes it often impossible to derive an explicit pricing formula due to the dependence of the payoff on the realized path of the underlying security. Therefore, one has to use a numerical procedure involving simulation or finite-difference methods. It is the latter that is the focus of this paper where we primarily demonstrate how parallel computing can effectively be used in pricing "backward-starting fixed-strike" Asian options that are averaged in continuous time. Consider a stock whose price changes according to the geometric Brownian motion

* Corresponding author. Tel.: +90 533 575 1674; fax: +90 212 285 7073.
E-mail addresses: halis.sak@boun.edu.tr (H. Sak), sozekici@ku.edu.tr (S. Özekici), ilkay@be.itu.edu.tr (İ. Boduroğlu).
URL: www.ilkay.net (İ. Boduroğlu).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2006.11.002

$$dS_t = S_t r\,dt + S_t \sigma\,dW_t$$

where W_t is a Wiener process or standard Brownian motion. There is also a zero coupon bond that is continuously traded at constant interest rate r . If Q is the so-called risk-neutral probability measure or the measure for which the discounted stock process $e^{-rt}S_t$ is a martingale, then given the filtration (or observed history) F_t until time t, the arbitrage-free price of the "backward-starting fixed-strike" Asian call option with maturity T is

$$C_t^a = E_Q\left[e^{-r(T-t)}\left(\frac{1}{T}\int_0^T S_u\,du - K\right)^+ \middle| F_t\right] \tag{1}$$

where K is the strike price of the option and we set $y^+ = \max\{y, 0\}$.

There has been many different approaches to price an Asian option using (1). Rogers and Shi [15], Alziary et al. [1] propose to solve a "one-state-variable" partial differential equation (PDE) after applying a change of numeraire in (1). Defining the new numeriare

$$x_t = \left[K - (1/T)\int_0^t S_u\,du\right] \Big/ S_t$$

the pricing equation is of the form $C_t^a = S_t \widehat{C}^a(x_t, t)$ where $\widehat{C}^a$ satisfies the PDE

$$\left(-\frac{1}{T} - rx\right)\widehat{C}_x^a(x,t) + \widehat{C}_t^a(x,t) + \frac{1}{2}\sigma^2 x^2 \widehat{C}_{xx}^a(x,t) = 0 \tag{2}$$

with the boundary condition

$$\widehat{C}^a(x,T) = x^- = \min\{-x, 0\}$$

which holds in the domain $\{(x,t) : -\infty < x < +\infty, 0 \leqslant t \leqslant T\}$. Here, $\widehat{C}_x^a(x,t) = \partial \widehat{C}^a(x,t)/\partial x$, $\widehat{C}_{xx}^a(x,t) = \partial^2 \widehat{C}^a(x,t)/\partial x^2$, and $\widehat{C}_t^a(x,t) = \partial \widehat{C}^a(x,t)/\partial t$. This PDE does not necessarily have an explicit solution, but it can be numerically evaluated using simulation or numerical finite-difference methods. However, if $K - (\int_0^t S_u\,du/T) < 0$, or equivalently $x_t = x \leqslant 0$, at time t then the explicit solution is known to be

$$\widehat{C}^a(x,t) = \frac{1}{Tr}(1 - e^{-r(T-t)}) - e^{-r(T-t)}x \tag{3}$$

for all $x \leqslant 0$. Using this result, Alziary et al. [1] consider (2) on $\{(x,t) : x \geqslant 0, 0 \leqslant t \leqslant T\}$ with (3) as the boundary condition at $x = 0$.

For the European call option, the corresponding PDE is

$$-rx\widehat{C}_x^e(x,t) + \widehat{C}_t^e(x,t) + \frac{1}{2}\sigma^2 x^2 \widehat{C}_{xx}^e(x,t) = 0 \tag{4}$$

with the boundary condition

$$\widehat{C}^e(x,T) = (1-x)^+$$

which holds in the domain $\{(x,t) : x \geqslant 0, 0 \leqslant t \leqslant T\}$.

Eq. (4) has, of course, the following explicit solution:

$$\widehat{C}^e(x,t) = \Phi(d_1) - xe^{-r(T-t)}\Phi(d_1 - \sigma\sqrt{T-t})$$

with

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\ln\left(\frac{1}{xe^{-r(T-t)}}\right) + \frac{1}{2}\sigma\sqrt{T-t}$$

where Φ is the cumulative standard normal distribution function. The only difference in (2) and (4) is the $1/T$ coefficient, which is a result of the "arithmetic average nature" of Asian options.

Monte Carlo simulation is used by [4,11,3] in order to price Asian options. The use of numerical finite-difference PDE methods starts with Schwartz et al. [17,5], who introduced the application of explicit and implicit

methods on the standard Black–Scholes equation. Courtadon [7] proposes a more accurate finite-difference approximation for the valuation of options; namely, the Crank–Nicolson (or mixed) finite-difference method. Hull and White [10] suggest a modification in the application of the explicit finite-difference method. Kemna and Vorst [11] establish a PDE with two state variables (the stock price and the average) which characterize the price of a "fixed-strike" Asian option. However, the pricing of Asian options is more complicated by using two state variables because of extra boundary conditions and computationally difficult numerical methods. Rogers and Shi [15] use the Numerical Algorithms Group's (NAG) routine (D03PAF) to solve the PDE. NAG develops and provides software to solve complex mathematical problems. On the other hand, Alziary et al. [1] use explicit finite-difference method to solve the same parabolic PDE. The use of parallel computing in option pricing using simulation is implemented in [6] in order to price discretely sampled Asian options.

In this paper, we solve the PDE by using both explicit and Crank–Nicolson's implicit finite-difference methods. We focus on algorithms designed for implementing these computations in parallel where we use Message Passing Interface functions to make the communications between parallel-processors in our parallel codes. Parallel implementation of the explicit finite-difference method is an easier task in contrast to the implicit one where one solves a system of linear equations. In our case, the system of linear equations has tridiagonal structure. Stone [19] proposes the recursive doubling algorithm as an efficient parallel algorithm for the solution of a tridiagonal system of linear equations. Wang [21] introduces the partition algorithm for the same problem, parallel factorization algorithm is proposed in [2] and improved by Mattor et al. [14]. A unified graph model to represent most of the algorithms, such as cyclic elimination, cyclic reduction and recursive doubling algorithms is proposed by Lin [13].

In our implementation on Asian option pricing, we compare the algorithms designed to work on MPPs and assess the efficiency of parallel computing. In this comparison, our criterion is the wall-clock times of algorithms and accuracy of the computed values. As the accuracy check, we use the percent error in European option pricing if we had used the same finite-spacings used in Asian option pricing as in [1]. Another check is provided by comparison of the prices with the prices evaluated in [15,22,12].

In Sections 2 and 3, we discuss the application of explicit and Crank–Nicolson's implicit finite-difference methods as well as algorithms designed to accomplish these computations in parallel. A detailed comparison of these methods and their parallel algorithms are presented in Section 4 using the accuracy and efficiency criteria. The computational results are given in the Appendix.

2. Explicit finite-difference method

We apply the same numerical approach to solve the one-state variable PDE as in [1] by first reducing the unbounded domain to a bounded one by a change of variables in the PDE. Although Alziary et al. [1] applies only the explicit finite-difference method, we apply both explicit and Crank–Nicolson's implicit finite-difference methods using parallel computing. However, the comparison of the accuracy of computed prices using different methods in the pricing of the Asian option which does not have an explicit pricing formula creates a problem. In this regard, Alziary et al. [1] implements the same change of numeriare approach to find the corresponding PDE for a European call option for which there is an explicit analytic pricing formula. Then, the PDE is numerically solved and the error in Asian option pricing is approximated by the error in European option pricing. This approximation of the error is somewhat justified because of the similarity in the PDEs. We also use this approach in our accuracy check and compute the error in European option pricing using the same finite-spacings for Asian option.

Alziary et al. [1] define a new state variable $y = e^{-x}$ and apply the change of variables to turn (2) into

$$\check{C}^a_t(y,t) + \frac{1}{2}\sigma^2(\ln y)^2 y^2 \check{C}^a_{yy}(y,t) + \left[\left(\frac{1}{T} - r\ln y\right)y + \frac{1}{2}\sigma^2 y(\ln y)^2\right]\check{C}^a_y(y,t) = 0 \tag{5}$$

with the following boundary conditions:

$$\check{C}^a(y,T) = 0, \quad \check{C}^a(1,t) = \frac{1}{Tr}(1 - e^{-r(T-t)}), \quad \check{C}^a(0,t) = 0$$

defined on the new bounded domain $\{(y,t) : 0 \leqslant y \leqslant 1, 0 \leqslant t \leqslant T\}$.

The corresponding PDE for the European call option after the change of variables is

$$\check{C}^e_t(y,t)+\frac{1}{2}\sigma^2(\ln y)^2y^2\check{C}^e_{yy}(y,t)+\left[-ry\ln y+\frac{1}{2}\sigma^2y(\ln y)^2\right]\check{C}^e_y(y,t)=0$$

with the boundary conditions

$$\check{C}^e(y,T)=(1+\ln y)^+,\quad \check{C}^e(1,t)=1,\quad \check{C}^e(0,t)=0$$

defined on domain $\{(y,t):0\leqslant y\leqslant 1,0\leqslant t\leqslant T\}$.

In the implementation of the explicit finite-difference method, we choose N increments (Δt) and M increments (Δy) of the variables t and y such that $N\Delta y=1$ and $M\Delta t=T$. We define $t_i=i\Delta t$ and $y_j=j\Delta y$ and the approximation of $\check{C}^a_{ij}=\check{C}^a(j\Delta y,i\Delta t)$. Replacing the partial derivatives with the estimations, we obtain from (5)

$$\check{C}^a_{i-1,j}=P_{j,j-1}\check{C}^a_{i,j-1}+P_{j,j}\check{C}^a_{i,j}+P_{j,j+1}\check{C}^a_{i,j+1} \tag{6}$$

for $j=1,\ldots,N-1$ where

$$P_{j,j-1}=\frac{1}{2}\sigma^2(\ln y_j)^2y_j^2\frac{\Delta t}{\Delta y^2}$$

$$P_{j,j}=1-\sigma^2(\ln y_j)^2y_j^2\frac{\Delta t}{\Delta y^2}-\left[\left(\frac{1}{T}-r\ln y_j\right)y_j+\frac{1}{2}\sigma^2y_j(\ln y_j)^2\right]\frac{\Delta t}{\Delta y}$$

$$P_{j,j+1}=\frac{1}{2}\sigma^2(\ln y_j)^2y_j^2\frac{\Delta t}{\Delta y^2}+\left[\left(\frac{1}{T}-r\ln y_j\right)y_j+\frac{1}{2}\sigma^2y_j(\ln y_j)^2\right]\frac{\Delta t}{\Delta y}$$

with the boundary conditions $\check{C}^a_{M,j}=0$ for $0\leqslant j\leqslant N$, and $\check{C}^a_{i-1,0}=0$, $\check{C}^a_{i-1,N}=\frac{1}{Tr}(1-\mathrm{e}^{-r(T-(i-1)\Delta t)})$. We apply (6) for $i=M$ to 1.

Furthermore, Alziary et al. [1] state that if σ and r are $\sigma^2\leqslant r$, the solution of (6) approaches to the solution of (5) as $\Delta y\to 0$ and $\Delta t\to 0$ provided that the stability condition

$$1-\frac{\sigma^2}{e^2}\frac{\Delta t}{\Delta y^2}-\frac{1}{T}\frac{\Delta t}{\Delta y}>0 \tag{7}$$

is satisfied.

2.1. Parallel implementation of the explicit method

It is clear that parallel computing could be extremely useful in the implementation of computational methods that require a huge number of computations. Massively Parallel computing can be defined as simultaneous usage of computers connected by a network in computing the solution of a problem. Parallel computing can speed up the process of computing by distributing the data to parallel-processors and executing the same or different instructions on these computers. This is the most common type of parallel computing known as "Multiple Instruction Multiple Data". Another advantage of parallel computing is the decrease obtained in the size of the problem by the distribution of data to parallel-processors so that it is possible to solve larger problems. This also has an impact on the reduction of wall-clock times in computing the solution of such problems. In parallel programming, we use Message Passing Interface (MPI) subroutines to provide the communication between parallel-processors. Parallel-processors exchange data by sending and receiving messages. This exchange should be cooperative, such as a send message should have a matching receive massage. To provide the parallelism between processors is the responsibility of the programmer.

We now present an algorithm to solve (6) using parallel computing with p processors. Recall that t and y axis are divided into M and N infinitesimal intervals of equal length respectively. The indices for the processors goes in the order of $P_0,P_1,\ldots,P_{p-1}$ and $\check{C}^a(i,j)$ stands for the relative price of the Asian option at i'th t column and j'th y row of the specified processor.

Parallel computing algorithm for the explicit method.

Step 1: Partition the y axis into p parallel-processors all over the time columns, so that each processor has N/p mesh points at each time column except processor 0 (P_0), which has $N/p+1$. Moreover, it is obvious from (6) that the unknown price at $(i-1,j)$ will be dependent on prices at $(i,j-1)$, (i,j) and $(i,j+1)$.

Step 2: Apply from $i=M$ to 1

P_0:

1- Send $\check{C}^a(i,N/p)$ to P_1.
2- Receive $\check{C}^a(i,N/p+1)$ from P_1.
3- Calculate $\check{C}^a(i-1,j)$ for $j=1$ to N/p using (6).

P_{p-1}:

1- Send $\check{C}^a(i,1)$ to P_{p-2}.
2- Receive $\check{C}^a(i,0)$ from P_{p-2}.
3- Calculate $\check{C}^a(i-1,j)$ for $j=1$ to $N/p-1$ using (6).

P_k ($k=1$ to $p-2$):

1- Send $\check{C}^a(i,1)$ to P_{k-1}.
2- Receive $\check{C}^a(i,0)$ from P_{k-1}.
3- Send $\check{C}^a(i,N/p)$ to P_{k+1}.
4- Receive $\check{C}^a(i,N/p+1)$ from P_{k+1}.
5- Calculate $\check{C}^a(i-1,j)$ for $j=1$ to N/p using (6).

3. Implicit finite-difference method

Although the explicit method is computationally simple, it has one serious drawback. The intervals Δt and Δy should be necessarily very small to attain reasonable accuracy. Crank and Nicolson [8] proposed and used a method that reduces the total amount of calculation and is valid (i.e., convergent) for all values of Δt and Δy. Replacing the partial derivatives with the estimations using Crank–Nicolson implicit finite-difference method, (5) becomes

$$(A_j^a - B_j^a)\check{C}^a_{i-1,j-1} - \left(\frac{1}{\Delta t} + 2A_j^a\right)\check{C}^a_{i-1,j} + (A_j^a + B_j^a)\check{C}^a_{i-1,j+1} = D^a_{i,j} \tag{8}$$

for $j=1,\ldots,N-1$ where

$$A_j^a = \frac{1}{4(\Delta y)^2}\sigma^2(\ln y_j)^2(y_j)^2$$

$$B_j^a = \frac{1}{4\Delta y}\left[\left(\frac{1}{T} - r\ln y_j\right)y_j + \frac{1}{2}\sigma^2 y_j(\ln y_j)^2\right]$$

$$D^a_{i,j} = (-A_j^a + B_j^a)\check{C}^a_{i,j-1} + \left(-\frac{1}{\Delta t} + 2A_j\right)\check{C}^a_{i,j} - (A_j^a + B_j^a)\check{C}^a_{i,j+1}$$

with the same boundary conditions as in the explicit finite-difference method. We apply (8) for $i=M$ to 1.

Note that (8) is a system of linear equations at a time column i. The relative price at time column $i-1$ is not dependent on only the relative prices at time column i as in the explicit method, but also on the two relative prices at time column $i-1$. Because of the boundary conditions, there are $N-1$ variables to be solved at each time column. There are many algorithms to solve systems of linear equations which can be categorized under three main headings: Gaussian Elimination, LU Decomposition and Iterative Methods. Our analysis will focus on the first two since our numerical analysis show they are quite efficient. Implementation of classical Iterative Methods, Jacobi, Gauss–Seidel and Successive-Over Relaxation methods can be found in [16].

3.1. Gaussian elimination method

We implement the Gaussian Elimination algorithm described in [18]. The system of linear equations in (8) can be shown in the following tridiagonal matrix form:

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & & & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} \check{C}_{i-1,1} \\ \check{C}_{i-1,2} \\ \check{C}_{i-1,3} \\ \cdot \\ \cdot \\ \check{C}_{i-1,N-2} \\ \check{C}_{i-1,N-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdot \\ \cdot \\ d_{N-2} \\ d_{N-1} \end{bmatrix} \tag{9}$$

where

$$a_j = A_j - B_j \quad (j = 2, 3, \ldots, N-1)$$
$$b_j = -\left(\frac{1}{\Delta t} + 2A_j\right) \quad (j = 1, 2, \ldots, N-1)$$
$$c_j = A_j + B_j \quad (j = 1, 2, \ldots, N-2)$$
$$d_j = D_{i,j} \quad (j = 2, 3, \ldots, N-2)$$

and

$$d_1 = (-A_1 + B_1)\check{C}_{i,0} + \left(-\frac{1}{\Delta t} + 2A_1\right)\check{C}_{i,1} - (A_1 + B_1)\check{C}_{i,2} - (A_1 - B_1)\check{C}_{i-1,0}$$
$$d_{N-1} = (-A_{N-1} + B_{N-1})\check{C}_{i,N-2} + \left(-\frac{1}{\Delta t} + 2A_{N-1}\right)\check{C}_{i,N-1} - (A_{N-1} + B_{N-1})\check{C}_{i,N} - (A_{N-1} + B_{N-1})\check{C}_{i-1,N}.$$

The first equation can be used to eliminate $\check{C}_{i-1,1}$ from the second equation, the new second equation can be used to eliminate $\check{C}_{i-1,2}$ from the third equation and so on, so that $\check{C}_{i-1,N-2}$ is eliminated from the last equation, giving one equation with only one unknown, $\check{C}_{i-1,N-1}$. Once this is determined, the other unknowns $\check{C}_{i-1,N-2}, \check{C}_{i-1,N-3}, \ldots, \check{C}_{i-1,2}, \check{C}_{i-1,1}$ can be found in turn by back-substitution. An algorithm that implements this procedure is to set

$$\alpha_j = b_j - a_j\left(\frac{c_{j-1}}{\alpha_{j-1}}\right) \quad (j = 2, 3, \ldots, N-1)$$
$$S_j = d_j - a_j\left(\frac{S_{j-1}}{\alpha_{j-1}}\right) \quad (j = 2, 3, \ldots, N-1)$$
$$\check{C}_{i,j} = \frac{1}{\alpha_j}(S_j - c_j\check{C}_{i,j+1}) \quad (j = N-2, \ldots, 2, 1)$$

for with $\alpha_1 = b_1$, $S_1 = d_1$, and $\check{C}_{i,N-1} = S_{N-1}/\alpha_{N-1}$.

3.1.1. Parallel implementation of Gaussian elimination method

There are ingenious parallel algorithms designed to solve tridiagonal system of linear equations by using the Gaussian Elimination method. Cyclic Reduction and Block Partition algorithms are two of these. Since we concentrate on distributed memory systems, Cyclic Reduction algorithm is not practical to use as it requires too much data movement. We discuss and implement the Block Partition algorithm described in [21] which starts by dividing the general system ($A\check{C} = d$) given in (9) into p subsystems (i.e., number of processors) each consisting of $n = (N-1)/p$ consecutive rows. For convenience, we assume $(N-1)$ is a multiple of p. Then, the elimination can proceed simultaneously on all subsystems by elementary row transformations until A is finally diagonalized. We use the notation presented in [13] to make the algorithm more clear and codable. We only give the algorithm here and refer the reader to [16] for figures and details.

Block partition algorithm.

Step 1: Partition the general system into p subsystems. We use the same variables in all the processors to run a unified code on all processors. Obviously, their values are not the same. This can be done because of the fact that each processor has the knowledge of only its data.
Step 2: Simultaneously apply the Gaussian elimination scheme to each submatrix. It should be clear that even when there is a change in the submatrix form, we do not change the variables. This strategy is used to avoid complexity. For all processors,

$$a_j = -a_{j-1}\frac{a_j}{b_{j-1}} \quad (j = 2, 3, \ldots, n)$$
$$b_j = b_j - c_{j-1}\frac{a_j}{b_{j-1}} \quad (j = 2, 3, \ldots, n)$$
$$d_j = d_j - d_{j-1}\frac{a_j}{b_{j-1}} \quad (j = 2, 3, \ldots, n).$$

Step 3: Simultaneously apply the Gaussian elimination scheme to each submatrix. For all processors except P_{p-1},

$$a_j = a_j - a_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-2, n-1, \ldots, 1)$$
$$c_j = -c_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-2, n-1, \ldots, 1)$$
$$d_j = d_j - d_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-2, n-1, \ldots, 1)$$

and for P_{p-1},

$$a_j = a_j - a_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-1, n-2, \ldots, 1)$$
$$c_j = -c_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-1, n-2, \ldots, 1)$$
$$d_j = d_j - d_{j+1}\frac{c_j}{b_{j+1}} \quad (j = n-1, n-2, \ldots, 1)$$

Step 4: Simultaneously apply the Gaussian elimination scheme to each submatrix. However, there have to be send–receive relationships between processors to apply the scheme. Moreover, here we introduce new variables such as b_0 to refer to the variable b_n of previous processor.

P_0:
1. Send b_n, c_n and d_n to P_1

P_{p-1}:
1. Receive b_0, c_0 and d_0 from P_{p-2}

Other processors (P_k):
1. Receive b_0, c_0 and d_0 from P_{k-1}
2. Send b_n, c_n and d_n to P_{k+1}

For all processors except P_0,

$$b_0 = b_0 - a_1\frac{c_0}{b_1}$$
$$c_0 = -c_1\frac{c_0}{b_1}$$
$$d_0 = d_0 - d_1\frac{c_0}{b_1}$$

P_0:
1. Receive b_n and d_n from P_1

P_{p-1}:
1. Send b_0 and d_0 to P_{p-2}

Other processors (P_k) :
1. Send b_0 and d_0 to P_{k-1}
2. Receive b_n and d_n from P_{k+1}.

Step 5: Apply the Gaussian elimination scheme to each submatrix sequentially. This stage is sequential since we have to wait until the values to be used in the next processor are determined. In addition, there have to be send–receive relationships between processors to apply the Gaussian elimination scheme. For P_0, do nothing. For P_1,

$$a_n = 0$$
$$b_n = b_n - c_0 \frac{a_n}{b_0}$$
$$d_n = d_n - d_0 \frac{a_n}{b_0}$$
Send b_n, d_n to P_2

For P_{p-1},

Receive b_0, d_0 from P_{p-2}
$$a_n = 0$$
$$b_n = b_n - c_0 \frac{a_n}{b_0}$$
$$d_n = d_n - d_0 \frac{a_n}{b_0}$$

and for other processors P_k,

Receive b_0, d_0 from P_{k-1}
$$a_n = 0$$
$$b_n = b_n - c_0 \frac{a_n}{b_0}$$
$$d_n = d_n - d_0 \frac{a_n}{b_0}$$
Send b_n, d_n to P_{k+1}

Step 6: Apply the Gaussian elimination scheme to each submatrix sequentially. This stage is also sequential since we have to wait until the values to be used in the previous processor are determined. In addition, there have to be send–receive relationships between processors to apply the Gaussian elimination scheme. For P_{p-1}, do nothing. For P_{p-2},

$$c_0 = 0$$
$$d_0 = d_0 - d_n \frac{c_0}{b_n}$$
Send d_0 to P_{p-3}

For P_0,

Receive d_n from P_1

and for other processors P_k,

Receive d_n from P_{k+1}

$c_0 = 0$

$d_0 = d_0 - d_n \dfrac{c_0}{b_n}$

Send d_0 to P_{k-1}

Step 7: Simultaneously apply the Gaussian elimination scheme to each submatrix. For all processors except P_0,

$a_j = 0 \quad (j = 1, 2, \ldots, n-1)$

$d_j = d_j - d_0 \dfrac{a_j}{b_0} \quad (j = 1, 2, \ldots, n-1)$

Step 8: Simultaneously apply the Gaussian elimination scheme to each submatrix. For all processors except P_{p-1},

$c_j = 0 \quad (j = 1, 2, \ldots, n-1)$

$d_j = d_j - d_n \dfrac{c_j}{b_n} \quad (j = 1, 2, \ldots, n-1)$

Step 9: The solution can now be computed by

$\check{C}_j = \dfrac{d_j}{b_j} \quad (j = 1, 2, \ldots, n)$

for all processors and this completes the algorithm description.

3.2. LU decomposition method

The principle in LU decomposition is to divide the system ($A\check{C} = d$) given in (9) into two separate systems. One of the matrices in the smaller systems is upper-triangular while the other one is lower-triangular. Instead of solving $A\check{C} = d$, we solve two smaller systems $Lz = d$ and $U\check{C} = z$ where L is the lower-triangular and U is the upper-triangular matrix. We implement [20] algorithm to solve the system of linear equations because of its efficiency. This algorithm gives the decomposition

$$L = \begin{bmatrix} 1 & & & & & & \\ bb_2 & 1 & & & & & \\ & bb_3 & 1 & & & & \\ & & \cdot & \cdot & & & \\ & & & \cdot & \cdot & & \\ & & & & bb_{N-2} & 1 & \\ & & & & & bb_{N-1} & 1 \end{bmatrix}$$

and

$$U = \begin{bmatrix} dd_1 & c_1 & & & & & \\ & dd_2 & c_2 & & & & \\ & & dd_3 & c_3 & & & \\ & & & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \\ & & & & & dd_{N-2} & c_{N-2} \\ & & & & & & dd_{N-1} \end{bmatrix}$$

where

$$bb_i = \frac{a_i}{dd_{i-1}}, \quad dd_i = b_i - bb_i c_{i-1}$$

for $i = 2, 3, \ldots, N-1$ with $bb_1 = 0$ and $dd_1 = b_1$. Finally, the solution of the subsystems $Lz = d$ and $U\check{C} = z$ are

$$z_i = d_i - bb_i z_{i-1} \quad (i = 2, 3, \ldots, N-1)$$
$$\check{C}_i = (z_i - c_i \check{C}_{i+1})/dd_i \quad (i = N-2, \ldots, 2, 1)$$

with $z_1 = d_1$ and $\check{C}_{N-1} = z_{N-1}/dd_{N-1}$.

3.2.1. Parallel implementation of LU decomposition method

Recursive Doubling and Parallel Factorization algorithms are parallel algorithms designed to solve tridiagonal system of linear equations by using LU decomposition. Duff and van der Vorst [9] state that the parallelism in Recursive Doubling algorithm is too fine-grained to be efficient. We discuss the Parallel Factorization algorithm which was first proposed by Amodio and Brugnano [2] and improved later by Mattor et al. [14].

Parallel factorization algorithm.

Step 1: Divide the system $(A\check{C} = d)$ given in (9) into p subsystems each consisting of $n = (N-1)/p$ consecutive rows. For convenience, we assume $(N-1)$ is a multiple of p. Denote the tridiagonal matrix formed at processor k by L_k.

Step 2: Three vectors x_k^{R}, x_k^{UH} and x_k^{LH} are defined as the solutions of the systems of equations given below. Apply these equations on all processors $k = 0, 1, \ldots, p-1$ to solve for x_k^{R}, x_k^{UH} and x_k^{LH}. The superscripts on the variable x_k stand for "particular" solution, "upper homogenous" solution and "lower homogenous" solution respectively.

$$L_k x_k^{\mathrm{R}} = d_k \tag{10}$$
$$L_k x_k^{\mathrm{UH}} = (-a_1, 0, 0, \ldots, 0)^{\mathrm{T}} \tag{11}$$
$$L_k x_k^{\mathrm{LH}} = (0, 0, 0, \ldots, -c_n)^{\mathrm{T}} \tag{12}$$

Step 3: The general solution x_k for processor $k = 0, 1, \ldots, p-1$ is the linear combination of x_k^{R}, x_k^{UH} and x_k^{LH} given by

$$x_k = x_k^{\mathrm{R}} + \xi_k^{\mathrm{UH}} x_k^{\mathrm{UH}} + \xi_k^{\mathrm{LH}} x_k^{\mathrm{LH}}$$

where ξ_k^{UH} and ξ_k^{LH} are the coefficients to be determined by solving the tridiagonal system of linear equations

$$\begin{bmatrix}
x_{1,n}^{\mathrm{LH}} & -1 & & & & & & & \\
-1 & x_{2,1}^{\mathrm{UH}} & x_{2,1}^{\mathrm{LH}} & & & & & & \\
 & x_{2,n}^{\mathrm{UH}} & x_{2,n}^{\mathrm{LH}} & -1 & & & & & \\
 & & -1 & x_{3,1}^{\mathrm{LH}} & x_{3,1}^{\mathrm{LH}} & & & & \\
 & & & x_{3,n}^{\mathrm{UH}} & x_{3,n}^{\mathrm{LH}} & -1 & & & \\
 & & & & \cdot & \cdot & \cdot & & \\
 & & & & & \cdot & \cdot & \cdot & \\
 & & & & & & \cdot & \cdot & -1 \\
 & & & & & & & -1 & x_{p-1,1}^{\mathrm{UH}}
\end{bmatrix}
\begin{bmatrix}
\xi_1^{\mathrm{LH}} \\ \xi_2^{\mathrm{UH}} \\ \xi_2^{\mathrm{LH}} \\ \xi_3^{\mathrm{UH}} \\ \xi_3^{\mathrm{LH}} \\ \cdot \\ \cdot \\ \xi_{p-2}^{\mathrm{LH}} \\ \xi_{p-1}^{\mathrm{UH}}
\end{bmatrix}
= -
\begin{bmatrix}
x_{1,n}^{\mathrm{R}} \\ x_{2,1}^{\mathrm{R}} \\ x_{2,n}^{\mathrm{R}} \\ x_{3,1}^{\mathrm{R}} \\ x_{3,n}^{\mathrm{R}} \\ \cdot \\ \cdot \\ x_{p-2,n}^{\mathrm{R}} \\ x_{p-1,1}^{\mathrm{R}}
\end{bmatrix}$$

with $\xi_1^{\mathrm{UH}} = \xi_{p-1}^{\mathrm{LH}} = 0$. Mattor et al. [14] employees the LU decomposition method to solve (10)–(12). Moreover, they present an algorithm to reduce the number of computations in the LU decomposition method by

exploiting the overlapping calculations and elements with zero values. Because of this important advantage, we also implement this algorithm by first setting

$$w_j = \frac{c_j}{b_j - a_j w_{j-1}}, \quad \gamma_j = \frac{d_j - a_j \gamma_{j-1}}{b_j - a_j w_{j-1}} \qquad (j = 2, 3, \ldots, n)$$

with $w_1 = c_1/b_1$ and $\gamma_1 = d_1/b_1$. Then,

$$x_j^{\mathrm{R}} = \gamma_j - w_j x_{j+1}^{\mathrm{R}}, \quad x_j^{\mathrm{LH}} = -w_j x_{j+1}^{\mathrm{LH}} \qquad (j = n-1, n-2, \ldots, 1)$$

with $x_n^{\mathrm{R}} = \gamma_n$ and $x_n^{\mathrm{LH}} = -w_n$. Finally,

$$w_j^{\mathrm{UH}} = \frac{a_j}{b_j - c_j w_{j+1}^{\mathrm{UH}}} \quad (j = n-1, n-2, \ldots, 1)$$

with $w_n^{\mathrm{UH}} = a_n/b_n$ and

Table 1
System parameter values for ASMA

System parameter	ASMA
OS	RedHat Linux 8.0
Linux kernel	2.4.18 – 14
C Compiler	gcc 3.2
Compiler options used	optimization level 2
MPI library used	MPICH 1.2.1 over TCP
Network Switch Environment	HP4000M Fast Ethernet Switch
Network Interface Card (NIC)	Intel Ether Express 100
Raw network bandwidth	100 Mb/s
MPI bandwidth	60 Mb/s
MPI latency	120 μs
Per-node memory	512 Mb DDR
Per-node CPU	AMD Athlon XP 2100+ (1733 MHz)
Node disk access type	local
Disk bandwidth (read–write)	(20 Mb/s)/(6 Mb/s)
Network load	Fluctuating
PL exponents ($\beta_{\mathrm{in}}/\beta_{\mathrm{out}}$)	2.1/2.7
Convergence criteria (Δ)	0.0001
Dampening factor (D)	0.85

Table 2
Computed Asian call option prices and bounds

σ	K	PDE	LB	RNI
0.05	95	11.094	11.094	11.094
	100	6.793	6.794	6.794
	105	2.748	2.744	2.744
0.10	90	15.399	15.399	15.398
	100	7.030	7.028	7.027
	110	1.410	1.413	1.413
0.2	90	15.643	15.641	15.641
	100	8.409	8.408	8.408
	110	3.554	3.554	3.554
0.30	90	16.514	16.512	16.512
	100	10.210	10.208	10.210
	110	5.729	5.728	5.730

$$x_j^{\mathrm{UH}} = -w_j^{\mathrm{UH}} x_{j-1}^{\mathrm{UH}} \quad (j = 2, 3, \ldots, n)$$

with $x_1^{\mathrm{UH}} = -w_1^{\mathrm{UH}}$.

This algorithm is extra efficient when the tridiagonal matrix does not change at different time columns as in our case. We can compute $w_j,\ x_j^{\mathrm{UH}},\ 1/(b_j + a_j w_{j-1}),\ x_j^{\mathrm{LH}}$ and use them for all time columns. Finally, note that we form the reduced system on all of the processors and each processor solves this system. To accomplish these, we broadcast the values needed to form the reduced matrix from each processor to all others. Then, to solve the reduced matrix, we use the LU decomposition method.

4. Comparison of the methods

In this section, we compare the methods introduced in the previous sections by executing their codes on Advanced System for Multi-computer Applications (ASMA) Linux PC cluster located in the Department

Table 3
Comparison of the explicit and implicit methods on a single-processor

σ	K	Explicit method					Implicit method				
		M	N	$\%E$	t_A	C^a	M	N	$\%E$	t_A	C^a
0.05	95	600	500	0.1555	0	10.9733	500	500	0.0010	1	11.0938
		1400	1000	0.0782	2	11.0289	1000	1000	0.0002	3	11.0941
		3500	2000	0.0395	11	11.0603	2000	2000	0.0001	14	11.0941
	100	600	500	0.1861	0	6.8105	500	500	0.0018	1	6.7872
		1400	1000	0.0977	2	6.7884	1000	1000	0.0005	3	6.7920
		3500	2000	0.0503	10	6.7856	2000	2000	0.0002	14	6.7937
	105	600	500	0.1274	1	3.2946	500	500	0.0280	1	2.6685
		1400	1000	0.0477	2	3.0890	1000	1000	0.0068	3	2.7253
		3500	2000	0.0190	10	2.9550	2000	2000	0.0016	14	2.7398
0.10	90	900	500	0.0908	1	15.2939	500	500	0.0013	1	15.3983
		2400	1000	0.0464	3	15.3423	1000	1000	0.0003	3	15.3986
		7500	2000	0.0236	22	15.3693	2000	2000	0.0000	14	15.3987
	100	900	500	0.1442	1	7.2616	500	500	0.0095	1	7.0103
		2400	1000	0.0712	4	7.1612	1000	1000	0.0022	3	7.0235
		7500	2000	0.0351	22	7.1033	2000	2000	0.0005	14	7.0267
	110	900	500	1.5373	0	2.004	500	500	0.0216	1	1.4218
		2400	1000	0.7800	4	1.7702	1000	1000	0.0064	3	1.4153
		7500	2000	0.3947	22	1.6235	2000	2000	0.0010	14	1.4141
0.20	90	2000	500	0.0422	1	15.6784	500	500	0.0017	1	15.6385
		7000	1000	0.0208	11	15.6613	1000	1000	0.0003	3	15.6409
		24,000	2000	0.0103	70	15.6517	2000	2000	0.0000	14	15.6416
	100	2000	500	0.2300	1	8.6952	500	500	0.0043	1	8.4039
		7000	1000	0.1155	11	8.5682	1000	1000	0.0008	3	8.4076
		24,000	2000	0.0578	70	8.4931	2000	2000	0.0001	14	8.4085
	110	2000	500	0.5806	2	3.9629	500	500	0.0035	1	3.5560
		7000	1000	0.2913	10	3.7834	1000	1000	0.0011	3	3.5556
		24,000	2000	0.1462	70	3.6766	2000	2000	0.0000	14	3.5556
0.30	90	3600	500	0.0697	3	16.6116	500	500	0.0010	1	16.5108
		14,000	1000	0.0347	23	16.5652	1000	1000	0.0002	3	16.5124
		52,000	2000	0.0173	153	16.5397	2000	2000	0.0000	14	16.5128
	100	3600	500	0.1652	3	10.4382	500	500	0.0022	1	10.2077
		14,000	1000	0.0826	23	10.3319	1000	1000	0.0004	3	10.2093
		52,000	2000	0.0413	153	10.2728	2000	2000	0.0001	14	10.2097
	110	3600	500	0.3026	2	6.0183	500	500	0.0015	1	5.7299
		14,000	1000	0.1512	21	5.8842	1000	1000	0.0004	3	5.7300
		52,000	2000	0.0757	153	5.8098	2000	2000	0.0001	14	5.7301

of Computer Engineering of Boğaziçi University. There are system parameters that may affect the outputs of the codes. These parameters and ASMA specifications for these parameters are given in Table 1.

Our criteria in this comparison are the wall-clock time of computers in executing the codes and accuracy of the price for European option. As stated earlier, to get information about how accurate we are in Asian option pricing, we also price the European option by the same approach used for the Asian option. All of the Asian and European option prices given in the Appendix are computed at $t = 0$ where we take the initial stock price $S_0 = 100$, the interest rate $r = 0.15$ and maturity is $T = 1$. We can check whether or not the per cent error in European option pricing is an indicator of the per cent error in pricing of Asian option by examining the Asian option prices in Table 2. Here, PDE stands for the partial differential solution of [22], LB is the lower bound on the price based on [15] and RNI represent the price determined by Recursive Numerical Integration in [12].

We first compare the explicit and Crank–Nicolson's implicit finite-difference methods on a single-processor. We prefer to use the LU decomposition method in the implementation of the Crank–Nicolson implicit

Table 4
Comparison of the explicit methods on a single-processor and on four parallel-processors

σ	K	Explicit method				Parallel explicit method				%E
		M	N	t_A	C^a	M	N	t_A	C^a	
0.05	95	600	500	0	10.9733	600	500	0	10.9733	0.1555
		1400	1000	2	11.0289	1400	1000	1	11.0289	0.0782
		3500	2000	11	11.0603	3500	2000	4	11.0603	0.0395
	100	600	500	0	6.8105	600	500	0	6.8105	0.1861
		1400	1000	2	6.7884	1400	1000	1	6.7884	0.0977
		3500	2000	10	6.7856	3500	2000	3	6.7856	0.0503
	105	600	500	1	3.2946	600	500	0	3.2946	0.1274
		1400	1000	2	3.0890	1400	1000	1	3.0890	0.0477
		3500	2000	10	2.9550	3500	2000	4	2.9550	0.0190
0.10	90	900	500	1	15.2939	900	500	0	15.2939	0.0908
		2400	1000	3	15.3423	2400	1000	2	15.3423	0.0464
		7500	2000	22	15.3693	7500	2000	8	15.3693	0.0236
	100	900	500	1	7.2616	900	500	0	7.2616	0.1442
		2400	1000	4	7.1612	2400	1000	2	7.1612	0.0712
		7500	2000	22	7.1033	7500	2000	7	7.1033	0.0351
	110	900	500	0	2.004	900	500	1	2.004	1.5373
		2400	1000	4	1.7702	2400	1000	1	1.7702	0.7800
		7500	2000	22	1.6235	7500	2000	8	1.6235	0.3947
0.20	90	2000	500	1	15.6784	2000	500	0	15.6784	0.0422
		7000	1000	11	15.6613	7000	1000	4	15.6613	0.0208
		24,000	2000	70	15.6517	24,000	2000	25	15.6517	0.0103
	100	2000	500	1	8.6952	2000	500	1	8.6952	0.2300
		7000	1000	11	8.5682	7000	1000	4	8.5682	0.1155
		24,000	2000	70	8.4931	24,000	2000	24	8.4931	0.0578
	110	2000	500	2	3.9629	2000	500	1	3.9629	0.5806
		7000	1000	10	3.7834	7000	1000	4	3.7834	0.2913
		24,000	2000	70	3.6766	24,000	2000	25	3.6766	0.1462
0.30	90	3600	500	3	16.6116	3600	500	1	16.6116	0.0697
		14,000	1000	23	16.5652	14,000	1000	10	16.5652	0.0347
		52,000	2000	153	16.5397	52,000	2000	51	16.5397	0.0173
	100	3600	500	3	10.4382	3600	500	2	10.4382	0.1652
		14,000	1000	23	10.3319	14,000	1000	8	10.3319	0.0826
		52,000	2000	153	10.2728	52,000	2000	53	10.2728	0.0413
	110	3600	500	2	6.0183	3600	500	2	6.0183	0.3026
		14,000	1000	21	5.8842	14,000	1000	8	5.8842	0.1512
		52,000	2000	153	5.8098	52,000	2000	53	5.8098	0.0757

finite-difference method for comparison purposes since it is more efficient than the Gaussian elimination method. The results are given in Table 3. Recall that M and N are the number of t and y intervals respectively over $[0,T]=[0,1]$ so that $\Delta t = 1/M$ and $\Delta y = 1/N$. Moreover, $\%E$ stands for the per cent error in European option pricing, t_A is the time spent on pricing of Asian option in seconds and, finally, C^a is the price computed for the Asian option for the given interest rate, volatility and strike price.

To compare the explicit and implicit methods, we need to make sure that the stability condition (7) is satisfied for the explicit method calculations. So, we keep N the same for both of the methods and look for M that makes the explicit method stable for N. Even though it is possible to apply the same M for the implicit method, this brings no improvement to the accuracy of the implicit method at the expense of time. Thus, we keep M and N the same for all implicit and iterative methods throughout the results.

As it can be seen from Table 3, the explicit method requires very large values of M for a fixed N as the volatility increases to satisfy the stability criterion. So, in the explicit method, we only store the data that belong to the last time column to avoid memory problems. It is clear that the implicit method is better at con-

Table 5
Comparison of the block partition and parallel factorization methods on four parallel-processors

σ	K	M	N	Block partition			Parallel factorization		
				$\%E$	t_A	C^a	$\%E$	t_A	C^a
0.05	95	500	500	0.0010	4	11.0938	0.0010	10	11.0938
		1000	1000	0.0002	5	11.0941	0.0002	23	11.0941
		2000	2000	0.0001	7	11.0941	0.0001	39	11.0941
	100	500	500	0.0018	1	6.7872	0.0018	13	6.7872
		1000	1000	0.0005	3	6.7920	0.0005	27	6.7920
		2000	2000	0.0002	9	6.7937	0.0002	40	6.7937
	105	500	500	0.0280	3	2.6685	0.0280	9	2.6685
		1000	1000	0.0068	3	2.7253	0.0068	22	2.7253
		2000	2000	0.0016	10	2.7398	0.0016	50	2.7398
0.10	90	500	500	0.0013	1	15.3983	0.0013	9	15.3983
		1000	1000	0.0003	3	15.3986	0.0003	22	15.3986
		2000	2000	0.0000	8	15.3987	0.0000	46	15.3987
	100	500	500	0.0095	3	7.0103	0.0095	11	7.0103
		1000	1000	0.0022	4	7.0235	0.0022	28	7.0235
		2000	2000	0.0005	5	7.0267	0.0005	36	7.0267
	110	500	500	0.0216	2	1.4218	0.0216	10	1.4218
		1000	1000	0.0064	4	1.4153	0.0064	26	1.4153
		2000	2000	0.0010	8	1.4141	0.0010	41	1.4141
0.20	90	500	500	0.0017	3	15.6385	0.0017	9	15.6385
		1000	1000	0.0003	5	15.6409	0.0003	22	15.6409
		2000	2000	0.0000	7	15.6416	0.0000	36	15.6416
	100	500	500	0.0043	2	8.4039	0.0043	15	8.4039
		1000	1000	0.0008	3	8.4076	0.0008	23	8.4076
		2000	2000	0.0001	11	8.4085	0.0001	45	8.4085
	110	500	500	0.0035	1	3.5560	0.0035	11	3.5560
		1000	1000	0.0011	3	3.5556	0.0011	22	3.5556
		2000	2000	0.0000	12	3.5556	0.0000	37	3.5556
0.30	90	500	500	0.0010	1	16.5108	0.0010	13	16.5108
		1000	1000	0.0002	4	16.5124	0.0002	20	16.5124
		2000	2000	0.0000	8	16.5128	0.0000	37	16.5128
	100	500	500	0.0022	3	10.2077	0.0022	10	10.2077
		1000	1000	0.0004	2	10.2093	0.0004	21	10.2093
		2000	2000	0.0001	10	10.2097	0.0001	46	10.2097
	110	500	500	0.0015	2	5.7299	0.0015	9	5.7299
		1000	1000	0.0004	3	5.7300	0.0004	20	5.7300
		2000	2000	0.0001	8	5.7301	0.0001	40	5.7301

verging to the accurate prices in the expense of time for small volatilities. Moreover, as volatility increases, the explicit method loses its competitive edge in speed because of the increase in M. If we check the compatibility of per cent error in European option pricing with per cent error in Asian option pricing by accepting the prices given in Table 2 as true prices, we can conclude that almost perfect fit is on hand especially for the implicit method.

Although the explicit method is much worse than Crank–Nicolson's implicit method, we can determine the performance of the parallel implementation of the explicit method over single-processor case. By interpreting Table 4, it is easy to conclude that parallel implementation of the explicit method with $p = 4$ processors is up to three times faster than the single-processor case for larger values of M and N. For smaller values of M and N, this statement is obviously biased because of time spend on send–receive relationships between processors.

Since the performance of the implicit method is superior over that of the explicit method, we want to focus on the efficiencies of Block Partition and Parallel Factorization algorithms in solving the system of

Table 6
Comparison of the block partition and parallel factorization methods on three parallel-processors

σ	K	M	N	Block partition			Parallel factorization		
				$\%E$	t_A	C^a	$\%E$	t_A	C^a
0.05	95	500	500	0.0010	1	11.0938	0.0010	1	11.0938
		1000	1000	0.0002	3	11.0941	0.0002	6	11.0941
		2000	2000	0.0001	10	11.0941	0.0001	9	11.0941
	100	500	500	0.0018	1	6.7872	0.0018	1	6.7872
		1000	1000	0.0005	4	6.7920	0.0005	4	6.7920
		2000	2000	0.0002	8	6.7937	0.0002	9	6.7937
	105	500	500	0.0280	1	2.6685	0.0280	3	2.6685
		1000	1000	0.0068	4	2.7253	0.0068	2	2.7253
		2000	2000	0.0016	6	2.7398	0.0016	11	2.7398
0.10	90	500	500	0.0013	0	15.3983	0.0013	1	15.3983
		1000	1000	0.0003	3	15.3986	0.0003	4	15.3986
		2000	2000	0.0000	11	15.3987	0.0000	6	15.3987
	100	500	500	0.0095	2	7.0103	0.0095	1	7.0103
		1000	1000	0.0022	2	7.0235	0.0022	3	7.0235
		2000	2000	0.0005	8	7.0267	0.0005	10	7.0267
	110	500	500	0.0216	1	1.4218	0.0216	1	1.4218
		1000	1000	0.0064	4	1.4153	0.0064	10	1.4153
		2000	2000	0.0010	7	1.4141	0.0010	14	1.4141
0.20	90	500	500	0.0017	1	15.6385	0.0017	1	15.6385
		1000	1000	0.0003	2	15.6409	0.0003	3	15.6409
		2000	2000	0.0000	11	15.6416	0.0000	8	15.6416
	100	500	500	0.0043	0	8.4039	0.0043	1	8.4039
		1000	1000	0.0008	4	8.4076	0.0008	5	8.4076
		2000	2000	0.0001	10	8.4085	0.0001	11	8.4085
	110	500	500	0.0035	0	3.5560	0.0035	3	3.5560
		1000	1000	0.0011	2	3.5556	0.0011	6	3.5556
		2000	2000	0.0000	6	3.5556	0.0000	10	3.5556
0.30	90	500	500	0.0010	2	16.5108	0.0010	2	16.5108
		1000	1000	0.0002	2	16.5124	0.0002	4	16.5124
		2000	2000	0.0000	6	16.5128	0.0000	10	16.5128
	100	500	500	0.0022	1	10.2077	0.0022	3	10.2077
		1000	1000	0.0004	3	10.2093	0.0004	5	10.2093
		2000	2000	0.0001	6	10.2097	0.0001	7	10.2097
	110	500	500	0.0015	3	5.7299	0.0015	1	5.7299
		1000	1000	0.0004	3	5.7300	0.0004	3	5.7300
		2000	2000	0.0001	7	5.7301	0.0001	17	5.7301

linear equations of the Crank–Nicolson implicit method. After examining Table 5, one can conclude that Block Partition and Parallel Factorization algorithms give the same prices as the single-processor implicit method given in Table 3. Although there is an improvement in time usage in the Block Partition algorithm for M and N values equal to 2000, Parallel Factorization algorithm performs worse than the single-processor implicit method for this level of M and N values using four parallel-processors. As M and N increases, the Parallel Factorization algorithm provides better efficiency in time usage over the Block Partition algorithm.

Although parameter changes in our tables for the same values of M and N for any method should not affect the time usage, there have been variations in durations of the runs. This is the result of the fact that the parallel methods highly depend on send–receive relationships between processors, and the speed is, therefore, quite sensitive to traffic on ASMA.

We also compared the performances of the Block Partition and Parallel Factorization algorithms using $p = 3$ three parallel-processor instead of four. The results are given in Table 6. If we compare their time usages, the Block Partition algorithm on three parallel-processor is a little bit faster than the four parallel-processor case for M and N equal to 2000. Moreover, time usage of the Parallel Factorization algorithm drops drastically and gets close to the Block Partition algorithm's value. Time usages of Block Partition and Parallel Factorization algorithms executed on three parallel-processor are approximately half of the single-processor implementation of Crank–Nicolson's implicit method using the LU Decomposition method given in Table 3.

We conclude that Crank–Nicolson's implicit finite-difference method is better than the explicit finite-difference method in pricing Asian options. Moreover, the LU decomposition method is better than the Gauss elimination method on a single-processor. Three parallel-processor implementation of Block Partition and Parallel Factorization algorithms' time usages are approximately half of the single-processor implementation of the Crank–Nicolson implicit method to attain reasonable accuracy. Furthermore, our results indicate that the per cent error in European option pricing is a very good estimator of the error in Asian option pricing by using the same pricing methodology because they have the similar PDE systems.

References

[1] B. Alziary, J.P. Décamps, P.F. Koehl, A PDE approach to Asian options: analytical and numerical evidence, Journal of Banking and Finance 21 (1997) 613–640.
[2] P. Amodio, L. Brugnano, Parallel factorizations and parallel solvers for tridiagonal linear systems, Linear Algebra and its Applications 172 (1992) 347–364.
[3] P. Boyle, M. Broadie, P. Glasserman, Monte Carlo methods for security pricing, Journal of Economic Dynamics and Control 21 (1997) 1267–1321.
[4] P. Boyle, D. Emanuel, Options on the General Mean, Working Paper, University of British Columbia, Vancouver, BC, 1980.
[5] M.J. Brennan, E.S. Schwartz, Finite difference method and jump processes arising in the pricing of contingent claims, Journal of Financial and Quantitative Analysis 13 (1978) 461–474.
[6] F.O. Bunnin, Y. Guo, Y. Ren, J. Darlington, Design of high performance financial modelling environment, Parallel Computing 26 (2000) 601–622.
[7] G. Courtadon, A more accurate finite difference approximation for the valuation of options, Journal of Financial and Quantitative Analysis 17 (1982) 301–330.
[8] J. Crank, P. Nicolson, A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type, Proceedings of the Cambridge Philsophical Society 43 (1947) 50–67.
[9] I.S. Duff, H.A. van der Vorst, Developments and trends in the parallel solution of linear systems, Parallel Computing 25 (1999) 1931–1970.
[10] J. Hull, A. White, Valuing derivative securities using the explicit finite difference method, Journal of Financial and Quantitative Analysis 25 (1) (1990) 87–100.
[11] A.G.Z. Kemna, A.C.F. Vorst, A pricing method for options based on average asset values, Journal of Banking and Finance 14 (1990) 113–129.
[12] T.W. Lim, Performance of recursive integration for pricing European-style Asian options, Working Paper, Department of Statistics and Applied Probability, National University of Singapore, Singapore, 2002.
[13] H.X. Lin, A unifying graph model for designing parallel algorithms for tridiagonal systems, Parallel Computing 27 (2001) 925–939.
[14] N. Mattor, T.J. Williams, D.W. Hewett, Algorithm for solving tridiagonal matrix problems in parallel, Parallel Computing 21 (1995) 1769–1782.
[15] L.C.G. Rogers, Z. Shi, The value of an asian option, Journal of Applied Probability 32 (1995) 1077–1088.

[16] H. Sak, Parallel computing in Asian option pricing, MS Thesis, Boğaziçi University, İstanbul, 2003.
[17] E.S. Schwartz, The valuation of warrants: implementing a new approach, Journal of Financial Economics 4 (1977) 79–93.
[18] G.D. Smith, Numerical Solution of Partial Differential Equations, Oxford University Press, Oxford, 1978.
[19] H.S. Stone, An efficient parallel algorithm for the solution of tridiagonal system of linear equations, Journal of the Association for Computing Machinery 20 (1973) 27–38.
[20] L.H. Thomas, Elliptic problems in linear difference equations over a network, Technical Report, Watson Scientific and Computational Laboratory, Columbia University, New York, 1949.
[21] H.H. Wang, A parallel method for tridiagonal equations, ACM Transactions on Mathematical Software 7 (1981) 170–183.
[22] R. Zvan, P. Forsyth, K. Vetzal, Robust numerical methods for PDE models of asian options, Journal of Computational Finance 1 (1997) 39–78.