A Parallel Algorithm for Random Walk Construction with Application to the Monte Carlo Solution of Partial Differential Equations

Abdul Yousef

Abstract—Random walks are widely applicable in statistical and scientific computations. In particular, they are used in the Monte Carlo method to solve elliptic and parabolic partial differential equations (PDE’s). This method holds several advantages over other methods for solving PDE’s as it solves problems with irregular boundaries and/or discontinuities, gives solutions at individual points, and exhibits great parallelism. However, the generation of each random walk in the Monte Carlo method has been done sequentially because each point in the walk is derived from the preceding point by moving one grid step along a randomly selected direction. In this paper, we present a parallel algorithm for the random walk generation in regular as well as irregular regions. The algorithm is based on parallel prefix computations, and takes \( O \left( \frac{L}{n} \log n \right) \) time, where \( L \) is the length of the random walk, and \( n \) is the number of processors. The communication structure of the algorithm is shown to ideally fit on a hypercube of \( n \) nodes.

Index Terms—Hypercube, Monte Carlo method, parallel algorithms, parallel prefix, partial differential equations, random walks.

I. INTRODUCTION

The Monte Carlo method has been studied and used to solve elliptic and parabolic partial differential equations (PDE) [8], [9], [14]. It has several advantages over other methods, such as solving problems with irregular boundaries and/or discontinuities, giving solutions at single points independently from the solutions at other points, and allowing for greater parallelism.

The greater amount of parallelism is derived from the fact that the solutions at different points are independent, panning the way to independent processes that can be executed in parallel. Moreover, the solution at each point consists of first evaluating a “primary estimator” (to be defined later) along each of many random walks in a grid, and then averaging these primary estimators. The random walks are independent and therefore constructable in parallel. Such parallel algorithms have been studied [1], and various computer architectures for their execution have been proposed [2], [3], [14].

In the above outline of the Monte Carlo method of solving PDE’s, the whole process is parallel except for the sequential construction of each random walk and the corresponding computation of the primary estimator. A random walk is constructed sequentially by starting at some initial point in a grid and moving in randomly generated directions until a boundary point is reached. Intuitively, the random walk construction seems inherently sequential because every point in the walk can be generated only if its preceding point is known.

The primary focus of this paper is the development of a parallel algorithm for constructing a random walk in rectangular and nonrectangular grids. The parallel algorithm will be derived by showing that random walk construction reduces in principle to prefix computation, which is a well-studied problem [4], [10]–[12]. The derived parallel algorithm significantly improves on the time to construct a random walk. Precisely, it reduces the time from linear to logarithmic (in the length of the walk).

II. THE MONTE CARLO METHOD AND ITS INHERENT PARALLELISM

Let

\[
A U_{xx} + 2 B U_{xy} + C U_{yy} + D U_x + E U_y + F = 0
\]

be a PDE defined over a region \( \Delta \) with boundary \( \gamma \). The factors \( A, B, C, D, E \) and \( F \) and the unknown function \( U \) are functions of \( x, y \) and possibly the time variable \( t \).

The Monte Carlo method is used to solve the following two problems:

A. The Elliptic PDE Problem:

\[
U \cdot A \cdot B \cdot C \cdot D \cdot E \quad \text{and} \quad F \quad \text{are} \quad \text{time-independent and} \quad B^2 - 4AC < 0 \quad \text{on} \quad \Delta. \quad \text{The problem is to solve} \quad (1) \quad \text{subject}
\]
Fig. 1. A grid point $P_0$ and its five relevant neighbors.

to the boundary condition

$$U(x, y) = \phi(x, y) \quad \text{if } (x, y) \in \gamma.$$  \hfill \(\text{(2)}\)

B. The parabolic PDE problem:

\begin{align}
U_{t} & = A_{B}C_{D}E \quad \text{and } F \text{ are time-dependent and } B^2 - AC = 0 \text{ on } \Delta. \text{ The problem is to solve (1) subject to} \\
\text{Boundary Condition} & : (x, y, t) = \phi(x, y, t) \quad \text{if } (x, y) \in \gamma \quad \text{and } t = 0 \quad \text{and (2)} \quad \text{and (3)} \quad \text{and (4)} \\
\text{Initial Condition} & : U(x, y, 0) = g(x, y) \quad \text{if } (x, y) \in \Delta.
\end{align}

The region $\Delta$ is divided into a regular grid of size $h$. Each point $P_i$ of the grid (except the boundary points) has 5 neighbors $P_{i-1}, P_{i+1}, P_{i-k}, P_{i+k}$ as depicted in Fig. 1. We denote by $h$, the direction along which we move from $P_0$ to $P_i$, where $i = 1, 2, 3, 4, 5$. A random number generator (RNG) generates random directions (i.e., $\theta_1, \theta_2, \ldots, \theta_5$). A random walk starting at $P_0$ is constructed by moving away from $P_0$ following directions generated by RNG until an absorbing point is hit. A point is said to be absorbing if it is a boundary point in the elliptic case, while in the parabolic case a point is absorbing if it is either a boundary point or a point reached at a certain specified time.

Let $r(P) = 2(A-B+C) + h(E+D)$, where $A, B, C, D, E$, and $F$ are evaluated at $(x, y)$, the coordinates of the point $P$. Let also $W_i$ be a random walk starting at $P_0$ and ending at a boundary point $Q_i$, and

$$Z(W_i) = \phi(Q_i) + h \sum_{P_j \in W_i} \frac{F(P_j)}{r(P_j)}.$$  \hfill \(\text{(5)}\)

The Monte Carlo solution of the elliptic equation (1) at point $P_0$, subject to condition (2), consists of generating a number of random walks $W_1, W_2, \ldots, W_N$, all starting at $P_0$ and ending at some boundary points $Q_1, Q_2, \ldots, Q_N$, respectively. Afterwards, $Z(W_i)$ is evaluated for all $i = 1, 2, \ldots, N$. Finally, $U(P_0)$ is approximated by

$$\theta = \frac{1}{N} \sum_{i=1}^{N} Z(W_i).$$  \hfill \(\text{(6)}\)

$Z(W_i)$ is called the primary estimator of $U(P_0)$, and $\theta$ the secondary estimator. For the proof that $\theta$ is a good approximation of $U$, see [14].

For the parabolic case, where $U$ and the coefficients of (1) are time dependent, the time scale is discretized into equal units of length $k$ (i.e., $t_s = sk$, $s \geq 0$), and $U(P_0), \phi(P_0), A, B, C, D, E, F$ and $r(P_0)$ at time $t_s$ are denoted $U_s(P_0), \phi_s(P_0), A_s, B_s, C_s, D_s, E_s, F_s, r_s(P_0)$, respectively. A random walk $W_i$ in this case is constructed as before except that $W_i$ is started at $P_0$ at time $t_s = sk$, and after each step (following a new direction), the time is decreased by one unit. The walk $W_i$ is finished if either a boundary point is reached (at some time $t_s$) or the time runs out (after $s$ steps), whichever comes first.

In this case, the primary estimator $Z(W_i)$ of $U_i(P_0)$ is

$$Z(W_i) = V_i(Q_0) + h^2 \sum_{j=0}^{s-1} F_{i-j}(P_j) r_{i-j}(P_j)$$  \hfill \(\text{(7)}\)

where $W_i$ is a random walk starting at $P_0$ and ending at some absorbing point $Q_i$, such that $Q_i$ is reached at time $t_s = sk$, and

$$V_i(Q_s) = \begin{cases} 
\phi_s(Q_s) & \text{if } Q_i \in \gamma \text{ and } a > 0 \\
0 & \text{if } a = 0 \text{ (i.e., } Q_i \text{ is reached at time } 0). 
\end{cases}$$

The Monte Carlo solution of the parabolic equation (1) at point $P_0$, at time $t_s$, subject to the conditions (3) and (4), consists of generating $N$ random walks $W_1, W_2, \ldots, W_N$, evaluating the $Z(W_i)$'s and averaging them as in the elliptic case.

It can be clearly seen that the random walks $W_1, W_2, \ldots, W_N$ are independent, and that $Z(W_1), Z(W_2), \ldots, Z(W_N)$ can be computed independently (i.e., in parallel). This inter-walk parallelism has been studied in [2], [3], and [14]. It is also clear to see that $U$ can be computed at different points independently and thus concurrently.

The third area for parallelism is the construction of each random walk $W$ and the computation of $Z(W)$ along with it. We refer to this process as the random walk computation (RWC). Although RWC seems inherently sequential, this paper will parallelize it.

III. PARALLEL CONSTRUCTION OF RANDOM WALKS

To parallelize the random walk construction we first need a number of independent random number (i.e., direction) generators that generate a sequence of random numbers simultaneously. Assume we have $n - 1$ independent random number generators running on $n - 1$ processors (the choice of $n - 1$ as opposed to $n$ will be justified later). The problem can be stated as follows:

Given a grid, a point $P_0$ in the grid, and a sequence of $n - 1$ random directions $d_1, d_2, \ldots, d_{n-1}$ generated by the $n - 1$ random number generators, construct in parallel the random walk consisting of the sequence of points $P_0, P_1, P_2, \ldots, P_{n-1}$ such that $P_i$ is the $i$th neighbor of $P_{i-1}$ for $i = 1, 2, \ldots, n - 1$.

For simplicity, assume first that the grid is an $m \times k$ rectangular grid labeled in such a way that the grid point $(i, j)$ is in the $i$th row and $j$th column, where $(0, 0)$ is in the bottom leftmost point. The case where the grid is not rectangular will be treated later.

A move along a direction $d$ can be viewed as a translation $T_d$ for some vector $A = (a, b)$ such that $T_d(i, j) = (i + a, j + b)$. The translations corresponding to the five directions are: $T_{(0,1)}$ for the move to the east direction, $T_{(0,-1)}$ for the move to the west direction, $T_{(1,0)}$ for the move to the north direction, $T_{(-1,0)}$ for the move to the south direction, and $T_{(-1,1)}$ for the move to the southeast direction.

Given an initial grid point $P_0$ and $n - 1$ random directions $d_1, d_2, \ldots, d_{n-1}$, the corresponding translations $T_{d_1}, T_{d_2}, \ldots, T_{d_{n-1}}$ can be simply found and the points $P_1, P_2, \ldots, P_{n-1}$ are derived as follows:

$$P_i = T_{d_{i-1}}(P_{i-1}) = P_{i-1} + A_{i-1}, \quad i = 1, 2, \ldots, n - 1,$$

where $P_0$ is the initial grid point and $A_0 = 0$ (i.e., $P_0$ is the origin).

Given an initial grid point $P_0$ and $n - 1$ random directions $d_1, d_2, \ldots, d_{n-1}$, the corresponding translations $T_{d_1}, T_{d_2}, \ldots, T_{d_{n-1}}$ can be simply found and the points $P_1, P_2, \ldots, P_{n-1}$ are derived as follows:

$$P_i = T_{d_{i-1}}(P_{i-1}) = P_{i-1} + A_{i-1}, \quad i = 1, 2, \ldots, n - 1,$$

where $P_0$ is the initial grid point and $A_0 = 0$ (i.e., $P_0$ is the origin).
boundary crossing. A point, \( P_r = (p, q) \) is a boundary point simply if \( p = 0, p = m = 1, q = 0 \) or \( q = k = 1 \), but what is required is to determine, in a parallel setting, the first boundary point and to discard all the remaining points after it. If there is no boundary crossing and \( P_{r-1} \) is not a boundary point, the algorithm has to be repeated until a boundary point is reached. The details of this algorithm are elaborated next.

The algorithm makes a heavy use of the procedure PAR-PREFIX, which does the prefix computation on \( n \) processors in parallel taking \( O(\log n) \) computation time, and, if run on a hypercube of \( n \) processors, it takes \( O(\log n) \) communication time. A similar procedure, called SCAN, is implemented on the Connection Machine [5]. Section III-A will present the procedure PAR-PREFIX. Section III-B will describe the parallel (RWC) for the elliptic PDE’s on rectangular grids. The case of nonrectangular grids is shown to be a slight variation of the rectangular case and is addressed in Section III-C. Afterwards, the necessary modifications to handle parabolic PDE’s are discussed in Section III-D.

A. The PAR-PREFIX Algorithm

The procedure PAR-PREFIX performs the parallel prefix computation for any associative operator. Specifically, the prefix problem is to compute the values of \( X_0, X_0 \diamond X_1, X_0 \diamond X_1 \diamond X_2, \ldots, X_0 \diamond X_1 \diamond \cdots \diamond X_{n-1} \), given the values \( X_0, X_1, \ldots, X_{n-1} \). The operator \( \diamond \) is an arbitrary associative binary operation such as scalar addition (+), vector addition (+), the minimum operator (MIN), the Boolean OR, and so on. The integer \( n \) can be assumed to be a power of 2. The parallel prefix problem is to compute the values above in parallel.

The parallel prefix problem has received much attention [4], [10]–[12], and various VLSI circuit implementations have been proposed [12], [13], [15]. For completeness, we will present in this paper an optimal parallel algorithm for prefix computation. The algorithm communication structure will be shown to ideally fit on a hypercube network of \( n \) nodes. This algorithm will be later generalized in Section IV to construct random walks in multidimensional grids.

The algorithm for PAR-PREFIX is best explained first as a recursive algorithm of one basis step, two recursive calls, and one combining step. Let \( X_{i,j} \) denote \( X_0 \circ X_1 \circ \cdots \circ X_i \). Initially processor \( p_e \) has \( X_i \). At the end of the algorithm \( p_e \) will have the value \( X_0 \) and \( X_{0,n-1} \). To understand the basis step of the recursive algorithm, assume there are two processors only, that is, \( n = 2 \). Then the algorithm proceeds as follows. \( p_{e0} \) sends \( X_0 \) to \( p_{e1} \) and \( p_{e1} \) sends \( X_1 \) to \( p_{e0} \). Afterwards, every processor computes \( X_{0,1} := X_0 \circ X_1 \). Thus, \( p_{e0} \) has now \( X_{0,0} \) (which is \( X_0 \)), \( p_{e1} \) has \( X_{0,1} \), and both \( p_e \)'s have \( X_0 \).

To understand the recursive part, assume that the processors are divided into halves. The first half consists of processors \( p_{e0}, p_{e1}, \ldots, p_{e(n-2)/2-1} \), and the second half consists of processors \( p_{e(n-2)/2}, p_{e(n-2)/2+1}, \ldots, p_{e(n-1)} \). The recursive part consists of performing PAR-PREFIX by the first half of processors on the data \( X_0, X_1, \ldots, X_{(n-2)/2-1} \), and also by the second half of processors on the data \( X_{(n-2)/2}, X_{(n-2)/2+1}, \ldots, X_{n-1} \). The effect of these two PAR-PREFIX’ s is that for every \( i = 0, 1, \ldots, (n/2) - 1 \), \( p_e \) has the value \( X_0 \), and the value \( X_{(n/2)-i} \). The combining step is next performed so that every processor \( p_{e1} \), for \( i = 0, 1, \ldots, n - 1 \), will have the values \( X_0 \), and \( X_{0,n-1} \). This step is accomplished as follows. Every processor \( p_{e1} \) in the first half sends the value \( X_{0,n-2}/2 \) to \( p_{e1+1+2}\ldots+2 \) in the second half. Similarly, every processor \( p_{e2}, p_{e3}, \ldots, p_{e(n-2)/2} \) in the second half sends the value \( X_{0,n-2}/2 \) to \( p_{e1} \) in the first half. Afterwards, every processor \( p_e \) in the second half computes \( X_{0,1} := X_{0,n-2}/2 \circ X_{0,n-2}/2 \). Finally, every \( p_e \), in both halves computes \( X_{0,n-1} := X_{0,n-2}/2 \circ X_{0,n-2}/2 \). By this combining step, every \( p_e \) has \( X_0 \) and \( X_{0,n-1} \).

This algorithm can be implemented recursively in \( \log n \) stages as follows. In the first stage, every pair of processors \( p_{e1} \) and \( p_{e1+1} \) do the same on their respective data as is done in the basis step of the recursive algorithm explained above. At stage \( i \), the \( n \) processors are divided into \( n/2^i \) independent subsystems \( I^i \), where \( I^i \) is the set of \( i \)'s of labels in \( j^2, (j + 1)^2 - 1 \] such that every processor \( p_e \) in the first half has the values \( X_{j^2}, X_{j^2+1}, \ldots, (j + 1)^2 - 1 \). Each subsystem \( I^i \) is in turn divided into two halves of processors \( j^2, j^2 + 2^i - 1 \) and \( j^2 + 2^i, (j + 1)^2 - 1 \) such that every processor \( p_e \) in the first half of \( j^2, j^2 + 2^i - 1 \) and every processor \( p_e \) in the second half has the values \( X_{j^2}, X_{j^2+1}, X_{j^2+2}, X_{j^2+3}, \ldots, X_{j^2+2^i-1} \). In stage \( i \), the processors in these two halves perform the same job on their data as the two halves in the combining step in the last paragraph. The details of this job are presented in the first inner for-loop in the procedure Stage(i) below. This procedure implements the \( i \)th stage just explained.

To fully understand the working of Stage(i), the semantics of three special parallel language constructs in Stage(i) need to be specified. The first is of the form

\[
\text{for } j = m \text{ to } k \text{ proc; } \text{pardo proc; } \text{endfor}
\]

which means that the processes \( \text{proc1, proc2, \ldots, procn} \) run simultaneously.

The second is of the form: \( p_e \) does: \( S \); which means that processor \( p_e \) executes the statement \( S \). The third is of the form \( \text{Send} (p_{e1}, p_{e2}, \ldots) \); which means that processor \( p_{e1} \) sends the data value \( a \) to processor \( p_{e2} \).

Procedure Stage(i)

\[
\text{for } j = 0 \text{ to } (n/2^i) - 1 \text{ pardo } /i j \text{ denotes the subsystem } I^i; \text{ for } l = j^2 \text{ to } j^2 + 2^i - 1 \text{ pardo } /l \text{ ranges over the first half of } I^i; \\
\text{Send} (p_{e1}, X_{j^2}, X_{j^2+1}, \ldots, X_{j^2+2^i-1}, p_{e1+1+2}\ldots+2^i); \text{ Send} (p_{e2}, X_{j^2+2^i+1}, \ldots, X_{j^2+2^i+2^i+1}, p_{e1+1+2}\ldots+2^i); \text{ Send} (p_{e3}, \ldots); \text{ Send} (p_{en-1}, \ldots); \text{ Send} (p_{e1+1+2}\ldots+2^i); \text{ Send} (p_{e2+1+2}\ldots+2^i); \text{ Send} (p_{e3+1+2}\ldots+2^i); \ldots; \text{ Send} (p_{en-1+1+2}\ldots+2^i).
\]

\[
\text{for } l = j^2 \text{ to } (j + 1)^2 - 1 \text{ pardo }
\]

\[
\text{pardo does: } X_{j^2}, X_{j^2+1}, \ldots, X_{j^2+2^i-1}, X_{j^2+2^i+1}, X_{j^2+2^i+2^i+1}, \ldots, X_{j^2+2^i+2^i+1}, X_{j^2+2^i+2^i+2^i+1}, \ldots
\]

\[
\text{endfor}
\]

\[
\text{endfor}
\]

\[
\text{endfor}
\]

\[
\text{end}
\]

The full algorithm for PAR-PREFIX is a simple sequential for-loop executing stage 1, stage 2, \ldots, stage \( \log n \), as presented below.

Procedure PAR-PREFIX \( (X(0..n-1)) \)

\[
\text{begin for } i = 1 \text{ to } \log n \text{ do Stage(i); endfor}
\]

Communication and Complexity analysis of PAR-PREFIX: By a simple inspection of the procedure Stage, we observe that communication occurs between processor \( p_e \) and \( p_{e+1+2}\ldots+2^i \) for various values of \( i \) and \( j \) such that \( i = 1, 2, \ldots, \log n \) and \( j^2 \leq l \leq j^2 + 2^i - 1 - 1 \). When \( l \) is expressed as a binary number \( l \), it can be
seen that \( l_{i-1} \) is equal to 0 and that \( l + 2^{i-1} \) has the same binary representation as \( l \) except that bit \( l_{i-1} \) is complemented. That is, \( l \) and \( l + 2^{i-1} \) differ in only one bit. Consequently, if PAR-PREFIX is run on a hypercube system of \( n \) processors, every two processors that need to communicate will have a direct link between them. In other terms, the hypercube structure is an ideal structure for PAR-PREFIX. For more detail on parallel prefix implementation on the hypercube, see [7].

Note that when the operator 0 is scalar addition, vector addition, the minimum operator MIN, or the Boolean OR, PAR-PREFIX is referred to as ADD-PREFIX, VADD-PREFIX, MIN-PREFIX, or OR-PREFIX, respectively.

B. Parallel RWC for the Elliptic PDE’s

Given an initial grid point \( P_0 \) and \( n-1 \) directions corresponding to the translations \( T_{1,0}, T_{1,1}, \ldots, T_{n-1} \), the points \( P_0, P_1, \ldots, P_{n-1} \) of the random walk are found by calling VADD-PREFIX(A(0..n-1)), where \( A(0) = P_0 \) and \( A(i) = A(i) \) for \( i = 1, 2, \ldots, n - 1 \). Since \( P_i = P_0 + A_1 + A_2 + \cdots + A_i \), we conclude that \( P_i = A_0 \) and is hence computed by \( P_e \). At this point, the boundary checking test has to be performed.

Assume that the coordinates of \( P_e \) are \((a_i, b_i)\), or equivalently, that \( A_0 = (a_i, b_i) \). Each \( P_e \) will check if the point \( P_e \) is a boundary point and will also compute a flag \( f \) as follows:

Procedure Check-Boundary(\( P_e, f \)) /* done by \( P_e \) */
begin
if \((a_i = 0 \text{ or } b_i = n - 1) \text{ or } (b_i = 0 \text{ or } b_i = k - 1)\)
then
\( f := 1 \);
else
\( f := 2n; \text{ or any number } > n */
end

To determine the first boundary point in which all succeeding points are discarded, we determine the minimum of all the flags \( f_0, f_1, \ldots, f_{n-1} \). This can be accomplished by executing \( \text{MIN-PREFIX}(f(0..n-1)) \). The minimum \( m \) is \( f_0 \) and is available at every \( P_e \), at the end of MIN-PREFIX. If all the points are within boundary, then each \( f_i = 2n \) and hence the minimum is \( 2n \). If there is a boundary point, assume that \( P_i \) is the first boundary point. Therefore, \( f_i = l \), and for \( i < l, f_i = 2n > f_i \). For \( i > l, f_i \) is either \( i \) or \( 2n \) based on whether \( P_i \) is a boundary point or not. Therefore, whether \( i < l \) or \( i > l, f_i = l < f_i \). Thus, \( f_l \) is the minimum of all the \( f_i \)'s, and is equal to it. It follows that \( m \) is equal to \( 2n \) if all points are within boundary, while if there is a boundary point, \( m \) is equal to the index of the first boundary point, that is, \( P_m \) is the first boundary point.

After the boundary checking and the computation of the minimum \( m \) of the flags \( f_i \)'s, each \( P_e \) checks if \( m = 2n \) (recall that every \( P_e \) has the \( m \) after MIN-PREFIX). If \( P_e \) finds \( m = 2n \) at \( i \leq m \), then \( P_e \) computes the value \( x_i = (F(P_i)/r(P_i))h^2 \) because the point \( P_i \) is on the walk. If \( i > m \), the point \( P_i \) is after the first boundary point, in which case \( P_e \) sets \( x_i \) to 0 (so that it will not contribute to \( Z(W) \)).

Now if \( m \neq 2n \), then \( P_m \) is the first boundary point (i.e., the endpoint \( Q \) of the random walk \( W \) just generated). In this case, \( P_{m+1} \) sets \( x_{m+1} = x_m + o(P_m) \). Afterwards, the processors \( P_{m+1}, P_{m+2}, \ldots, P_{n-1} \) sum their \( x_i \)'s to form \( Z(W) \). This is accomplished by executing ADD-PREFIX(x(0..n-1)).

On the other hand, if \( m = 2n \), then no boundary point has been reached yet. The sum \( x_0 + x_1 + \cdots + x_{n-1} \) is computed using ADD-PREFIX(x(0..n-1)), and then stored in a temporary variable \( Z \) in \( P_0 \). Afterwards, \( P_0 \) sets the points \( P_0 \) to \( P_{n-1} \) (i.e., \( P_i := P_{n-1} \)), while all the other \( P_e \)'s clear all their variables, and the whole process of generating \( n - 1 \) random directions and finding new points is repeated. The "Z"-value of every new set of points is computed and added to the old Z variable. The process is repeated until a boundary point is reached.

These steps of the parallel construction of a random walk and the computation of the associated primary estimator are summarized in procedure PAR-RWC.

Procedure PAR-RWC(\( P_0 \))
begin
1. \( Z := 0; \)
2. for \( i = 1 \) to \( n - 1 \) pardo
\( P_e \) randomly generates a direction \( d \), and computes the corresponding \( A_i \).
endfor
3. VADD-PREFIX(A(0..n-1)); /*Find the n points \( P_0, P_1, \ldots, P_{n-1} */
4. for \( i = 0 \) to \( n - 1 \) pardo
\( P_e \) does : Check-Boundary(\( P_i, f_i \));
endfor
5. MIN-PREFIX(f(0..n-1)); /*\( m = f_{n-1} \) which is the minimum of all \( f_i \)'s is stored in every \( P_e */
6. for \( i = 0 \) to \( n - 1 \) pardo
\( P_e \) does : if \((m = 2n \text{ or } i \leq m)\) then \( x_i := (F(P_i)/r(P_i))h^2 \); else \( x_i := 0 \);
endfor
7. if \((m \neq 2n)\) then /* An absorbing point \( P\) is reached */
\( P_{m+1} \) does : \( x_{m+1} := x_m + o(P_m) \);
ADD-PREFIX(x(0..n-1));
\( Z := Z + x_{n-1} \);
return;
else /*No absorbing point has been reached*/ ADD-PREFIX(x(0..n-1));
\( Z := Z + x_\)next
Send \((P_{e-1}, P_{e-1}, P_{e-1})\);
\( P_0 \) does : \( P_i := P_{i-1} \);
goto 2; /* Repeat the process */
end

The execution time of PAR-RWC is \( O([L/n] \log n) \) on a hypercube architecture of \( n \) processors, where \( L \) is the length of the random walk. This is so because each goto-iteration takes \( O(\log n) \) time, and the number of goto-iterations is \( [L/n] \).

This ends the development of the parallel RWC algorithm for rectangular grids and elliptic PDE's. In the next two subsections, nonrectangular regions are handled and then the modifications needed to handle the parabolic PDE's are presented.

C. Handling Nonrectangular Regions

For the case of nonrectangular convex regions, the region is embedded in the smallest rectangular \( m \times k \) grid such that the boundary lines of the grid are tangent to the region where the upper/lower lines are horizontal. The two points of intersection between the region boundary and row \( p \) of the grid are recorded for each \( p \). The western intersection point takes the label of the grid point immediately to its west (denoted \( (p, W(p)) \)), and the eastern intersection point takes the label of the grid point immediately to
its east (denoted $(p, E(p))$). Similarly, the two points of intersection between the region boundary and column $q$ of the grid are recorded for each $q$. The northern intersection point takes the label of the grid point immediately to its north (denoted $(N(q), q)$), and the southern intersection point takes the label of the grid point immediately to its south (denoted $(S(q), q)$). The random walk generation PAR-RWC is the same as in the rectangular case except for boundary checking. A point $P = (p, q)$ is a boundary point if $p = N(q)$ or $S(q)$, or $q = E(p)$ or $W(p)$. Thus the Check-Boundary procedure becomes:

**Procedure Check-Boundary($P$, $f$)/* done by $pe$.*

```plaintext
begin
  if $a = N(b)$ or $a = S(b)$ or $b = E(a)$ or $b = W(a)$
  then
    $*$ $P$ is on boundary/*
    $f := f$;
  else
    $f := 2n; /*$ or any number $> n/*$
  end
```

To be able to execute this procedure, every $pe$ has to store $W(p)$, $E(p)$, $N(q)$, and $S(q)$ for all $0 \leq p \leq m - 1$ and $0 \leq q \leq k - 1$.

Thus the parallel RWC algorithm keeps its simplicity and speed, requiring only some additional storage for the intersection points between the region boundary and the grid.

**D. Parallel RWC for Parabolic PDE’s**

The parallel RWC algorithm for the parabolic case is very similar to the algorithm for the elliptic case. The only difference is the definition of absorbing points and the subsequent change needed to detect absorbing points. $Z(W)$ is also slightly modified.

At the outset of the algorithm, every $pe$ has a counter $T$ (for time) initialized to $s$. The algorithm finds the $n - 1$ points first in the same way as in the elliptic case. It also performs boundary checking as before. If a boundary point has been detected, say $P_{mn}$, and if $T - mn \geq 0$, then $P_{mn}$ is an absorbing point, and hence every $pe$, for $j \leq mn$, computes $x_j := (F_{mn}(P_{mn})/r_{mn}(P_{mn}))h^2$, and then $pe_{mn}$ sets $x_{mn} \leftarrow x_{mn} + \sigma_{mn}(P_{mn})$ as required to compute $Z(W)$ of (7). All remaining $pe$’s set their $x_j$’s to 0. $Z(W)$ is then computed by executing ADD-PREFIX as in the elliptic case.

If on the other hand, $T - mn < 0$ (i.e., time-out), then the absorbing point is $P_T$. In this case, only the $pe_j$’s where $j \leq T$ compute $x_j := (F_{j-1}(P_j)/r_{j-1}(P_j))h^2$, while all the remaining $pe$’s set their $x_j$’s to 0. Then $P_T$ performs $x_T := x_T + g(P_T)$. Afterwards, $Z(W)$ is computed using ADD-PREFIX($c_{\infty}(0, m - 1)$).

If $mn = 2n$ (i.e., no grid boundary point is reached) and if $T \leq n - 1$, then $P_T$ is an absorbing point and the algorithm does as in the preceding paragraph. However, if $T > n - 1$, then no absorbing point has been reached. In this case, the same computations are done as in the elliptic case (to accumulate $Z$), but before we repeat the algorithm with a new set of $n - 1$ random directions, the counter $T$ in each $pe$ is updated: $T := T - (n - 1)$. Afterwards, the algorithm is repeated until an absorbing point is reached.

As can be seen, the additional computations needed for the parabolic case take constant time. Consequently, the overall time for the parallel RWC for the elliptic or parabolic PDE’s is $O(\frac{n}{\log n})$.

**IV. RANDOM WALKS IN MULTIDIMENSIONAL GRIDS**

This section will address scaling up the parallel RWC algorithm to $m$-dimensional grids where $m \geq 2$. Recall that the $n$ points $P_0, P_1, \ldots, P_{n-1}$ of a random walk are generated by conducting a parallel prefix (of vector addition) on $n$ vectors $A_0, A_1, \ldots, A_{n-1}$, where $A_0 = P_0$ and $A_1$ is a vector that characterizes the direction to be followed in the grid to generate $P_1$ from $P_{-1}$. Each point $P_j$ is equal to $A_0 + A_1 + \cdots + A_{j-1}$.

In a $p_1 \times p_2 \times \cdots \times p_m$ grid of dimension $m$, the vector $A_j$ consists of $m$ numbers. Therefore, if the vector addition is to be done by single processors, it takes $m$ time steps. This forces the parallel prefix to take $O(m \log n)$ time on an $n$-node hypercube. However, this time requirement can be easily cut down by a factor of $m$ on an $m$-node hypercube by using the $m$-fold natural parallelism in the vector addition operation. This is explained below.

1. Divide the $m$-node hypercube into $n$ subcubes of $n$ nodes each. The labels of the nodes in each subcube agree in the log $n$ most significant bits. These log $n$ bits are taken to be the label of the subcube. View each subcube as a "hypernode". Due to the connectivity of hypercubes, these $n$ hypernodes form an $n$-node hypercube where between every two neighboring hypercubes there exist $m$ links (one link between every pair of corresponding nodes in the two hypercubes).

2. To do a parallel prefix on the vectors $A_0, A_1, \ldots, A_{n-1}$, each $A_i$ is stored in the hypercube of label $i$ such that the $j$th number of $A_i$ is stored in the $j$th position of $A_i$ (i.e., $p_{i,j}$, where $i \cdot j$ denotes the concatenation of the binary representations of $i$ and $j$). Perform the parallel prefix on the $n$-hypercube hypercube (as in Section III) treating each vector addition and each Send as an atomic operation executed by a hypercube.

3. The atomic vector addition of two vectors $(v_0, v_1, \ldots, v_{m-1})$ and $(w_0, w_1, \ldots, w_{m-1})$ is done in parallel by the $m$ nodes of the hypercube:

```plaintext
for $j = 0$ to $m - 1$
  $p_{i,j} := (i.e., \ j$th node of hypernode $i$ does $a_j + v_j$;
endfor
```

This clearly takes one time unit needed for one scalar addition.

4. Similarly, the atomic vector Send sends a vector $(v_0, v_1, \ldots, v_{m-1})$ from one hypernode $i$ to a neighboring hypernode $k$ in this parallel fashion:

```plaintext
for $j = 0$ to $m - 1$
  $p_{i,j}$ does:
    if $(y_j = 0 \text{ or } y_j = p_j - 1)$ then
      $c_j := 1; /*$ Hit the boundary in dimension $j$ */*
    else
      $c_j := 0;
    endif
  endfor
```

This takes one unit of communication time because there is a direct link between $pe_{i,j}$ and $pe_{k,j}$.

5. When the $n$ points $P_0, P_1, \ldots, P_{n-1}$ have been computed, boundary checking has to be conducted. Each hypernode $i$ takes advantage of its being an $m$-node hypercube to check if its point $P_i = (y_0, y_1, \ldots, y_{m-1})$ is a boundary point as follows:

```plaintext
for $j = 0$ to $m - 1$
  $p_{i,j}$ does:
    if $(c_{0,j} = 1)$ then
      $i := j; /*$ $P_i$ is a boundary point */*
    endif
```

OR-PREFIX($c_{0,m-1}$); /* computes the Boolean or of the $c_j$'s */
Afterwards, the \( n \)-node subcube consisting of the processors \( P_{i} \) for \( i = 0, 1, \ldots, n - 1 \) performs \( \text{MIN-PREFIX}(f_{0..n-1}) \) as in the previous section to check if a boundary point has been reached, and if so, to calculate the first boundary point. If no boundary point is reached, the process is repeated.

Steps 1–4 take \( O(\log n) \) time, and step 5 takes \( O(\log m) \) time. Thus, the process of computing a random walk of length \( L \) is \( O([L/n] \log m + \log n) = O([L/n] \log (mn)) \).

It should be noted that the random walk can also be computed on the \( mn \)-node hypercube using the approach of the previous section where each vector addition operation is done on a single processor (in \( m \) steps). The random walk construction takes \( O([L/mn] m \log (mn)) \) in this nonhypernode approach. Observe that for all \( L, n \) and \( m, [L/n] \leq [L/mn] m \). Also, when \( n < L < mn \), we have \( [L/n] < [L/mn] m \). Consequently, the hypernode approach is at least as fast as the nonhypernode approach. This makes the hypernode approach preferable.

V. CONCLUSIONS

We have presented in this paper a parallel algorithm for the construction of random walks and applied it to the Monte Carlo solution of elliptic and parabolic partial differential equations. The algorithm was shown to ideally fit on a hypercube structure. The algorithm is optimal in time and space when the region is a rectangular grid. It is also optimal in time when the region is irregular. This parallel construction of random walks offers great speedup in the solution of partial differential equations. It reduces the time of random walk construction from linear to logarithmic time in the length of the random walk.

The parallel random walk construction algorithm was also generalized to multidimensional grids and shown to execute efficiently on hypercubes.