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

Abdou Youssef

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 (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 $O(\lceil L/n\rceil \log n)$ 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, paving 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).

Manuscript received april 6, 1990; revised October 21, 1991.
The author is with the Department of Electrical Engineering and Computer Science, The George Washington University Washington, DC 20052.
IEEE Log Number 9205857.

After the parallel generation of the points of a random walk, one needs to check whether a boundary has been reached and, if so, to determine the first point at which the boundary is crossed because the succeeding points have to be discarded. With a novel use of parallel prefix, a second parallel algorithm for boundary checking and first-boundary point calculation is presented. This algorithm is also logarithmic in time.

Since random walks have many applications in statistical and scientific computations where the grid is multidimensional, the parallel algorithm for random walk construction is generalized to handle multidimensional grids. The paper will also show how to ideally map this generalized algorithm onto a hypercube system so that the computation time and the communication time are minimized.

A secondary focus of this paper is the application of parallel random walk construction to the Monte Carlo solution of PDE's. As was indicated earlier, the construction of random walks was the only remaining sequential segment of the Monte Carlo algorithm. With the new parallel algorithm for random walk construction, the time needed by the Monte Carlo method is significantly reduced if the method is run on a parallel computing system with enough processors. Clearly, no commercial system currently available has enough processors to take advantage of all the natural parallelism as well as the new parallelism in this problem. However, the advances in VLSI and computer architecture make the prospects for massively parallel systems promising. Systems with hundreds of thousands or even millions of processors may be a reality in the near future. The improved Monte Carlo algorithm will then run much faster on such massively parallel systems.

It should be pointed out that several techniques for solving PDE's have been used. Examples include the well-known finite difference method and finite element method. We pointed out at the outset the advantages of the Monte Carlo method. However, it is outside the scope of this paper to thoroughly compare the Monte Carlo method (with or without parallel random walk construction) to these other methods. Suffice it to say that whenever the use of the Monte Carlo method to solve PDE's (or any problem) is advantageous, and if enough processors are available, the parallel random walk construction should be used because it significantly improves the performance of the the Monte Carlo method.

This paper is organized as follows. The next section presents briefly the Monte Carlo method for PDE's and identifies the role of random walks. Section III develops a parallel algorithm for the random walk computation. Section IV generalizes the parallel random walk construction to grids of multiple dimensions. Conclusions are presented in Section V.

II. The Monte Carlo Method and its Inherent Parallelism

Let

$$AU_{xx} + 2BU_{xy} + CU_{yy} + DU_x + EU_y + F = 0 \qquad (1)$$

be a PDE defined over a region Δ with boundary γ. 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, A, B, C, D, E, and F are time-independent and $B^2 - AC < 0$ on Δ. The problem is to solve (1) subject

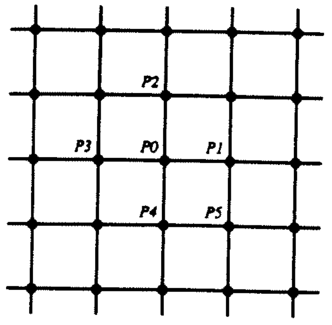

Fig. 1. A grid point P_0 and its five relevant neighbors.

to the boundary condition

$$U(x,y) = \phi(x,y) \quad if\ (x,y) \in \Gamma \tag{2}$$

B. The parabolic PDE problem:
U, A, B, C, D, E, and F are time-dependent and $B^2 - 4AC = 0$ on Δ. The problem is to solve (1) subject to

$$Boundary\ Condition: \quad U(x,y,t) = \phi(x,y,t) \quad if\ (x,y) \in \Gamma \tag{3}$$

$$Initial\ Condition: \quad U(x,y,0) = g(x,y) \quad if\ (x,y) \in \Delta. \tag{4}$$

The region Δ is divided into a regular grid of size h. Each point P_0 of the grid (except the boundary points) has 5 neighbors $P_1, P_2, \cdots, P_5$ as depicted in Fig. 1. We denote by δ_i 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., $\delta_1, \delta_2, \cdots, \delta_5$). 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

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

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 $W_1, W_2, \cdots, W_N$, all starting at P_0 and ending at some boundary points $Q_1, Q_2, \cdots, Q_N$, respectively. Afterwards, $Z(W_i)$ is evaluated for all $i = 1, 2, \cdots, N$. Finally, $U(P_0)$ is approximated by

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

$Z(W_i)$ is called the primary estimator of $U(P_0)$, and θ the secondary estimator. For the proof that θ 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 and $r_s(P_0)$, 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_α) or the time runs out (after s steps), whichever comes first.

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

$$Z(W_i) = V_\alpha(Q_i) + h^2 \sum_{j=0}^{s-\alpha} \frac{F_{s-j}(P_j)}{r_{s-j}(P_j)} \tag{7}$$

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 $t_\alpha = \alpha k$,

and

$$V_\alpha(Q_i) = \begin{cases} \phi_\alpha(Q_i) & \text{if } Q_i \in \Gamma \text{ and } \alpha \geq 0 \\ g(Q_i) & \text{if } \alpha = 0 \text{ (i.e., } Q_i \text{ is reached at time 0)} \end{cases}.$$

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 $W_1, W_2, \cdots, W_N$, evaluating the $Z(W_i)$'s and averaging them as in the elliptic case.

It can be clearly seen that the random walks $W_1, W_2, \cdots, W_N$ are independent, and that $Z(W_1), Z(W_2), \cdots, Z(W_N)$ 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 $d_1, d_2, \cdots, d_{n-1}$ generated by the $n - 1$ random number generators, construct in parallel the random walk consisting of the sequence of points $P_0, P_1, P_2, \cdots, P_{n-1}$ such that P_i is the d_ith neighbor of P_{i-1} for $i = 1, 2, \cdots, n - 1$.

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 ith row and jth 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_A for some vector $A = (a, b)$ such that $T_A(i, j) = (i, j) + A = (i + a, j + b)$. 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 $d_1, d_2, \cdots, d_{n-1}$, the corresponding translations $T_{A_1}, T_{A_2}, \cdots, T_{A_{n-1}}$ can be simply found and the points $P_1, P_2, \cdots, P_{n-1}$ are derived as follows: $P_1 = T_{A_1}(P_0) = P_0 + A_1$, $P_2 = T_{A_2}(P_1) = T_{A_2}T_{A_1}(P_0) = P_0 + A_1 + A_2$, and in general, $P_i = T_{A_i}(P_{i-1}) = T_{A_i} \cdots T_{A_2}T_{A_1}(P_0) = P_0 + A_1 + A_2 + \cdots + A_i$, for $i = 1, 2, \cdots, n - 1$. Finding these points is now a standard *parallel prefix computation* [4], [10]–[12] which can be solved in $O(\log n)$ time. However, further work has to be done to check boundary crossing. A point $P_i = (p, q)$ 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_{n-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, $X_0 \circ X_1$, $X_0 \circ X_1 \circ X_2, \cdots$, $X_0 \circ X_1 \circ \cdots \circ X_{n-1}$, given the values $X_0, X_1, \cdots, X_{n-1}$. The operator $\circ$ is any 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 $X_i \circ X_{i+1} \circ \cdots \circ X_j$. Initially processor pe_i has X_i. At the end of the algorithm pe_i will have the value $X_{0:i}$ and $X_{0:n-1}$. To understand the basis step of the recursive algorithm, assume there are 2 processors only, that is, $n = 2$. Then the algorithm proceeds as follows. pe_0 sends X_0 to pe_1 and pe_1 sends X_1 to pe_0. Afterwards, every pe computes $X_{0:1} := X_0 \circ X_1$. Thus, pe_0 has now $X_{0:0}$ (which is X_0), pe_1 has $X_{0:1}$, and both pe's have $X_{0:1}$.

To understand the recursive part, assume that the processors are divided into halves. The first half consists of processors $pe_0, pe_1, \cdots, pe_{n/2-1}$, and the second half consists of processors $pe_{n/2}, pe_{(n/2)+1}, \cdots, pe_{n-1}$. The recursive part consists of performing PAR-PREFIX by the first half of processors on the data $X_0, X_1, \cdots, X_{(n/2)-1}$, and also by the second half of processors on the data $X_{n/2}$, $X_{(n/2)+1}, \cdots, X_{n-1}$. The effect of these two PAR-PREFIX's is that for every $i = 0, 1, \cdots, (n/2) - 1$, pe_i has the value $X_{0:i}$ and the value $X_{0:(n/2)-1}$, and that for every $i = n/2, (n/2) + 1, \cdots, n - 1$, pe_i has the value $X_{(n/2):i}$ and the value $X_{(n/2):n-1}$. The combining step is next performed so that every processor pe_i, for $i = 0, 1, \cdots, n - 1$, will have the values $X_{0:i}$ and $X_{0:n-1}$. This step is accomplished as follows. Every processor pe_i in the first half sends the value $X_{0:(n/2)-1}$ to $pe_{i+(n/2)}$ in the second half. Similarly, every processor $pe_{i+(n/2)}$ in the second half sends the value $X_{(n/2):n-1}$ to pe_i in the first half. Afterwards, every pe_i in the second half computes $X_{0:i} := X_{0:(n/2)-1} \circ X_{(n/2):i}$. Finally, every pe_i in both halves computes $X_{0:n-1} := X_{0:(n/2)-1} \circ X_{(n/2):n-1}$. By this combining step, every pe_i has $X_{0:i}$ and $X_{0:n-1}$.

This algorithm can be implemented nonrecursively in $\log n$ stages as follows. In the first stage, every pair of processors pe_{2l} and pe_{2l+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 $(I_j)_{0 \leq j \leq (n/2^i)-1}$, where I_j is the set of pe's of labels in $[j2^i, (j+1)2^i - 1] = \{j2^i, j2^i + 1, j2^i + 2, \cdots, (j+1)2^i - 1\}$. Each subsystem I_j is in turn divided into two halves of processors $[j2^i, j2^i + 2^{i-1} - 1]$ and $[j2^i + 2^{i-1}, (j+1)2^i - 1]$ such that every pe_l in the first half has the values $X_{j2^i:l}$ and $X_{j2^i:j2^i+2^{i-1}-1}$ (computed in stage $i - 1$), and every pe_l in the second half has the values $X_{j2^i+2^{i-1}:l}$ and $X_{j2^i+2^{i-1}:(j+1)2^i-1}$. 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 ith 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

```
for $j$ = $m$ to $k$ pardo
    $proc_j$;
endfor
```

which means that the processes $proc_m, proc_{m+1}, \cdots, proc_k$ run simultaneously.

The second is of the form: pe_l **does**: S; which means that processor pe_l executes the statement S. The third is of the form **Send** (pe_l, a, pe_s); which means that processor pe_l sends the data value a to processor pe_s.

```
Procedure Stage($i$)
begin
  for $j = 0$ to $(n/2^i) - 1$ pardo   /*$j$ denotes the subsystem $I_j$*/
    for $l = j2^i$ to $j2^i + 2^{i-1} - 1$pardo / *$l$ ranges over the first
                                                    half of $I_j$*
      Send ($pe_l, X_{j2^i:j2^i+2^{i-1}-1}, pe_{l+2^{i-1}}$);
      Send ($pe_{l+2^{i-1}}, X_{j2^i+2^{i-1}:(j+1)2^i-1}, pe_l$);
      $pe_{l+2^{i-1}}$ does: $X_{j2^i:l+2^{i-1}} := X_{j2^i:j2^i+2^{i-1}-1} \circ$
                                        $X_{j2^i+2^{i-1}:l+2^{i-1}}$;
    endfor

    for $l = j2^i$ to $(j+1)2^i - 1$ pardo
      $pe_l$ does: $X_{j2^i:(j+1)2^i-1} := X_{j2^i:j2^i+2^{i-1}-1} \circ$
                                        $X_{j2^i+2^{i-1}:(j+1)2^i-1}$;
    endfor
  endfor
end
```

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

```
Procedure PAR-PREFIX ($X(0..n-1)$)
begin
  for $i$ = 1 to $\log n$ do
    Stage($i$);
  endfor
end
```

Communication and Complexity analysis of PAR-PREFIX: By a simple inspection of the procedure **Stage**, we observe that communication occurs between processor pe_l and $pe_{l+2^{i-1}}$ for various values of l and i such that $i = 1, 2, \cdots, \log n$ and $j2^i \leq l \leq j2^i + 2^{i-1} - 1$. When l is expressed as a binary number $l_{n-1} \cdots l_1 l_0$, 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 $\circ$ 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 $T_{A_1}, T_{A_2}, \cdots, T_{A_{n-1}}$, the points $P_0, P_1, \cdots, P_{n-1}$ 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 $i = 1, 2, \cdots, n-1$. Since $P_i = P_0 + A_1 + A_2 + \cdots + A_i$, we conclude that $P_i = A_{0:i}$ and is hence computed by pe_i. At this point, the boundary checking test has to be performed.

Assume that the coordinates of P_i are (a_i, b_i), or equivalently, that $A_{0:i} = (a_i, b_i)$. Each pe_i will check if the point P_i is a boundary point and will also compute a certain *flag* f_i as follows:

```
Procedure Check-Boundary($P_i, f_i$) /* done by $pe_i$*
begin
  if ($a_i = 0$ or $a_i = m - 1$) or ($b_i = 0$ or $b_i = k - 1$)
then
    /* $P_i$ is on boundary*/
    $f_i := i$;
  else
    $f_i := 2n$; /* or any number $> n$*/
end
```

To determine the first boundary point so that all succeeding points are discarded, we determine the minimum of all the flags $f_0, f_1, \cdots, f_{n-1}$. This can be accomplished by executing MIN-PREFIX($f(0..n-1)$). The minimum min is $f_{0:n-1}$ and is available at every pe_i 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_l is the first boundary point. Therefore, $f_l = l$, and for $i < l$, $f_i = 2n > f_l$. 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 $i > l$, $f_l = l < f_i$. Thus, f_l is the minimum of all the f_i's, and is equal to l. It follows that min is equal to $2n$ if all points are within boundary, while if there is a boundary point, min is equal to the index of the first boundary point, that is, P_{min} is the first boundary point.

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

Now if $min \neq 2n$, then P_{min} is the first boundary point (i.e., the endpoint Q of the random walk W just generated). In this case, pe_{min} sets x_{min} to $x_{min} := x_{min} + \phi(P_{min})$. Afterwards, the processors $pe_0, pe_1, \cdots, pe_{n-1}$ 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 $min = 2n$, 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 pe_0. Afterwards, pe_0 sets the points P_0 to P_{n-1} (i.e., $P_0 := P_{n-1}$), while all the other pe'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
      $pe_i$ randomly generates a direction $d_i$ and computes the
          corresponding $A_i$;
   endfor
3. VADD-PREFIX($A(0..n-1)$); /*Find the $n$ points
      $P_0, P_1, \cdots, P_{n-1}$*/
4. for $i = 0$ to $n - 1$ pardo
     $pe_i$ does : Check-Boundary($P_i, f_i$);
   endfor
5. MIN-PREFIX($f(0..n-1)$);
   /*$min = f_{0:n-1}$ which is the minimum of all $f_i$'s and is
     stored in every $pe$*/
6. for $i = 0$ to $n - 1$ pardo
     $pe_i$ does : if ($min = 2n$ or $i \leq min$) then
          $x_i := \frac{F(P_i)}{r(P_i)}h^2$;
        else
          $x_i := 0$;
        endif
   endfor
7. if ($min \neq 2n$) then /* An absorbing point $P_{min}$ is reached */
        $pe_{min}$ does : $x_{min} := x_{min} + \phi(P_{min})$;
        ADD-PREFIX($x(0..n-1)$);
        $Z := Z + x_{0:n-1}$;
         return;
   else /*No absorbing point has been reached*
        ADD-PREFIX($x(0..n-1)$);
        $Z := Z + x_{0:n-1}$;
        Send ($pe_{n-1}, P_{n-1}, pe_0$);
        $pe_0$ does : $P_0 := P_{n-1}$;
        goto 2; /* Repeat the process */
    endif
 end
```

The execution time of PAR-RWC is $O(\lceil L/n \rceil \log n)$ 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 $\lceil L/n \rceil$.

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($P_i$, $f_i$) /* done by $pe_i$,
                                        $P_i = (a_i, b_i)$*/
begin
  if $a_i = N(b_i)$ or $a_i = S(b_i)$ or $b_i = E(a_i)$ or $b_i = W(a_i)$
then
    /* $P_i$ is on boundary*/
    $f_i := i$;
  else
    $f_i := 2n$; /* 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_i 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_{min}, and if $T - min \geq 0$, then P_{min} is an absorbing point, and hence every pe_j, for $j \leq min$, computes $x_j := (F_{T-j}(P_j)/r_{T-j}(P_j))h^2$, and then pe_{min} sets x_{min} to $x_{min} + \sigma_{T-min}(P_{min})$ as required to compute $Z(W)$ of (7). All the remaining pe's set their x's to 0. $Z(W)$ is then computed by executing ADD-PREFIX as in the elliptic case.

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

If $min = 2n$ (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 $O(\lceil \frac{L}{n} \rceil \log n)$, whether the region is a rectangular grid or not, where L is the length of the random walk, and n is the number of processors.

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 $P_0, P_1, \cdots, P_{n-1}$ of a random walk are generated by conducting a parallel prefix (of vector addition) on n vectors $A_0, A_1, \cdots, A_{n-1}$, where $A_0 = P_0$ and A_i is a vector that characterizes the direction to be followed in the grid to generate P_i from P_{i-1}. Each point P_i is equal to $A_0 + A_1 + \cdots + A_{n-1}$.

In a $p_1 \times p_2 \times \cdots \times p_m$ grid of dimension m, the vector A_i 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 mn-node hypercube by using the m-fold natural parallelism in the vector addition operation. This is explained below.

1) Divide the mn-node hypercube into n subcubes of m 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 hypernodes there exist m links (one link between every pair of corresponding nodes in the two hypernodes).
2) To do a parallel prefix on the vectors $A_0, A_1, \cdots, A_{n-1}$, each A_i is stored in the hypernode of label i such that the jth number of A_i is stored in the jth pe of hypernode i (i.e., $pe_{i \bullet j}$, where $i \bullet j$ denotes the concatenation of the binary representations of i and j). Perform the parallel prefix on the n-hypernode hypercube (as in Section III) treating each vector addition and each **Send** as an atomic operation executed by a hypernode.
3) The atomic vector addition of two vectors $(u_0, u_1, \cdots, u_{m-1})$ and $(v_0, v_1, \cdots, v_{m-1})$ is done in parallel by the m nodes of the hypernode:

```
for $j = 0$ to $m-1$ pardo
   $pe_{i \bullet j}$ (i.e., the $j$th node of hypernode $i$) does : $u_j + v_j$;
endfor
```

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

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

```
for $j = 0$ to $m-1$ pardo
  Send ($pe_{i \bullet j}, v_j, p_{k \bullet j}$);
endfor
```

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

5) When the n points $P_0, P_1, \cdots, P_{n-1}$ 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 $P_i = (y_0, y_1, \cdots, y_{m-1})$ is a boundary point as follows:

```
for $j = 0$ to $m-1$      pardo
    $pe_{i \bullet j}$ does :      if ($y_j = 0$ or $y_j = p_j - 1$) then
                           $c_j := 1$; /* Hit the boundary
                                         in dimension $j$ */
                        else
                           $c_j := 0$;
                        endif
endfor
OR-PREFIX($c(0..m-1)$); /* computes the Boolean or of
the $c_j$'s*/
$pe_{i \bullet 0}$ does :           if ($c_{0:m-1} = 1$) then
                              $f_i := i$; /* $P_i$ is a
                           boundary point */
```

```
        else
                $f_c := 2n$;
        endif
```

Afterwards, the n-node subcube consisting of the processors $pe_{i,0}$'s for $i = 0, 1, \cdots, n-1$ performs 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 $O(\lceil L/n \rceil (\log m + \log n)) = O(\lceil L/n \rceil \log(mn))$.

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 $O(\lceil L/mn \rceil m \log(mn))$ in this nonhypernode approach. Observe that for all L, n and m, $\lceil L/n \rceil \leq \lceil L/mn \rceil m$. Also, when $n < L < mn$, we have $\lceil L/n \rceil < \lceil L/mn \rceil m$. 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.

References

[1] V. C. Bhavsar, "Some parallel algorithms for Monte Carlo solutions of partial differential equations," in *Advances in Computer Methods for Partial Differential Equations*, vol. 4, R. Vichnevestky and R. S. Stepleman, Eds. New Brunswick: IMACS, 1981, pp. 135–141.

[2] V. C. Bhavsar and V. V. Kantkar, "A multiple microprocessor system (MMPS) for the Monte Carlo solution of partial differential equations," in *Advances in Computer Methods for Partial Differential Equations*, vol. 2, R. Vichnevestky, Ed. New Brunswick: IMACS, 1977, pp. 205–213.

[3] V. C. Bhavsar and A. J. Padgaonkar, "Effectiveness of some parallel computer architectures for the Monte Carlo solution of partial differential equations," in *Advances in Computer Methods for Partial Differential Equations*, vol. 3, R. Vichnevestky and R. S. Stepleman, Eds. New Brunswick: IMACS, 1979, pp. 259–264.

[4] G. Bilardi and F. P. Preparata, "Size-time complexity of Boolean networks for prefix computations," *J. ACM*, vol. 36, no. 2, pp. 362–382, Apr. 1989.

[5] G. E. Blelloch, "Scans as primitive operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.

[6] J. H. Curtiss, "Sampling methods applied to differential and difference equations," in *Proc. Seminar Scientif. Computat.*, IBM 1949.

[7] Ö. Eğecioğlu, E. Gallopoulos, and Ç. Koç, "Fast and practical parallel polynomial interpolation," Tech. Rep. 646, Center for Supercomputing Research and Development, Univ. Illinois at Urbana-Champaign, Jan. 1987.

[8] J. H. Halton, "A retrospective and prospective survey of the Monte Carlo method," *SIAM Rev.*, vol. 12, Jan. 1970.

[9] J. M. Hammersly and D. C. Handscomb, *Monte Carlo Methods.* London, England: Methuen, 1964.

[10] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[11] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.*, vol. C-34, pp. 965–968, 1985.

[12] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM* vol. 27, pp. 831–838, 1980.

[13] S. Lakshmivarahan, C.-M. Yang, and S. K. Dhall, "On a class of optimal parallel prefix circuits with $(size + depth) = 2n - 2$ and $\lceil \log n \rceil \leq depth \leq (2\lceil \log n \rceil - 3)$," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 58–63.

[14] E. Sadeh and M. A. Franklin, "Monte Carlo solutions of partial differential equations by special purpose digital computers," *IEEE Trans. Comput.*, vol. C-23, pp. 389–397, Apr. 1974.

[15] M. Snir, "Depth-size trade-off for parallel prefix computation," *J. Algorithms*, vol. 7, pp. 185–201, 1986.