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

Abdou Youssef
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052

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 $O(\frac{L}{n}\log n)$ 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.

§1. Introduction

The Monte Carlo Method has been studied and used to solve elliptic and parabolic partial differential equations (PDE) [5], [6], [11]. It holds 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 great parallelism.

The great amount of parallelism is drawn from the fact that the solution at different points are independent, paving the way to indenpendent processes that can be executed in parallel. Moreover, the solution at each point consists of evaluating a "primary estimator" (to be defined later) along a large number of random walks, then averaging these values. The random walks are independent and

therefore constructable in parallel, and the estimations along the random walks can be computed in parallel as well. Such parallel algorithms have been studied [1], and various computer architectures for their execution have been proposed [2], [3], [12].

A much less obvious amount of parallelism can be introduced into the evaluation of the primary estimator along a random walk. It is less obvious because the random walk is constructed sequentially making the computation proportional to the length of the random walk.

In this paper we develop and study a parallel algorithm for the construction of random walks and along with it the evaluation of the primary estimator, reducing the time of this part of the solution from $O(L)$ to $O(\log L)$, where L is the length of the random walk. It should be noted that this parallel random walk generation algorithm has other applications in statistical and scientific computations where random walks and monte carlo methods are used.

This paper is organized as follows. The next section presents briefly the Monte Carlo Method for PDE's and points out all the possible areas of parallelism. Section 3 introduces the intra-walk parallelism and gives a parallel algorithm for the random walk computation. Conclusions are presented in section 4.

§2. The Monte Carlo Method and its Inherent Parallelism

Let

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

be a PDE and Δ a region 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 solve equation (1) subject 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 $B^2 - AC = 0$ on δ. The problem is solve equation (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, ... , P_5 as depicted in Figure 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., δ_1, δ_2, ... , δ_5). A random walk starting at P_0 is constructed by moving away from P_0 following directions generated by RNG till 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 parapolic case, it is absorbing if it is either a boundary point or a point reached at a certain specified time.

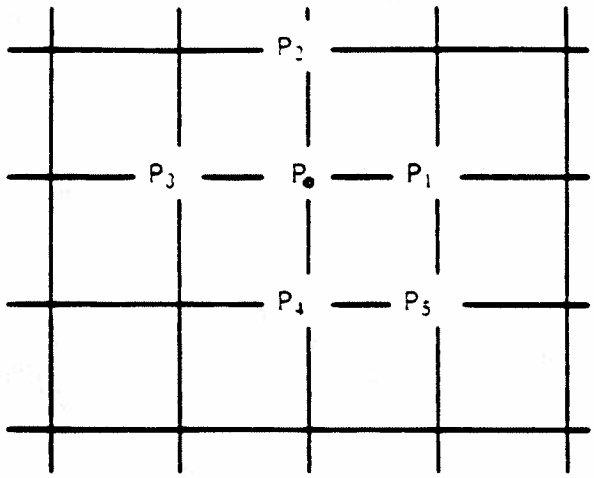

Figure 1

Let $r(P_0) = 2(A - B + C) + h(E + D)$, where A, B, C, D, E and F are evaluated at (x,y), the coordinates of the point P_0. 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, ... , W_N, all starting at P_0 and ending

at Q_1, Q_2, ... , Q_N, respectively. Afterwards, $Z(W_i)$ is evaluated for all $i = 1, 2, \ldots, N$. Finally, $U(P_0)$ is approximated by

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

$Z(W_i)$ is called the primary estimator of $U(P_0)$, and θ the secondary estimator. For the proof that this method yields a good approximation of U, see !num.

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 in this case is constructed as before except that W 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 is finished if either a boundary point is reached 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:

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

where Q_i is the last point of W reached at time $t_a = ak$, and

$$V_a(Q_i) = \begin{cases} \phi_a(Q_i) & \text{if } Q_i \in \gamma \text{ and } a \geq 0 \\ g(Q_i) & \text{if } a = 0 \text{ (i.e., } Q_i \text{ is reached at time 0, and may be a non-boundary point)} \end{cases}$$

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 W_1, W_2, ... , 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, ... , W_N are independent, and that $Z(W_1)$, $Z(W_2)$, ... , $Z(W_N)$ can be computed independently (i.e., in parallel). This inter-walk parallelism has been studied in [2], [3], [11]. 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 W and the computation of $Z(W)$ along with it. We refer to this process as the random walk computation (RWC). At first glance, the construction of a random walk seems inherently sequential since a current point has to

be known before the succeding point is found using a random direction generator. However, this paper will parallelize the construction of a random walk.

§3. 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 that starts at P_0 and moves away following the directions $d_1, d_2, \ldots, d_{n-1}$, consecutively. That is, the walk is a sequence of points $P_0, P_1, P_2, \ldots, P_{n-1}$ such that P_i is the d_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 handled later.

A move along a direction d can be viewed as a trnaslation 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 $d_1, d_2, \ldots, d_{n-1}$, the corresponding translations $T_{A_1}, T_{A_2}, \ldots, T_{A_{n-1}}$ can be simply found and the points $P_1, P_2, \ldots, P_{n-1}$ 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, $P_i = T_{A_i}(P_{i-1}) = T_{A_i} \ldots T_{A_2}T_{A_1}(P_0) = P_0 + A_1 + A_2 + \ldots + A_i$, for $i = 1, 2, \ldots, n-1$. Finding these points is now a standard *parallel prefix computation* [7] [8] [9] 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 is not a boundary point, the algorithm is repeated. The

details of this algorithms are elaborated next.

The algorithm makes a heavy use of the procedure SCAN 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 is implemented on the Connection Machine. Subsection 3.1 will present the procedure SCAN. Subsection 3.2 will describe the parallel (RWC) for the elliptic PDE's on rectangular grids. The case of non-rectangular grids is shown to be a slight variation of the rectangular case and is handled in Subsection 3.3. Afterwards, the necessary modifications to handle parabolic PDE's are discussed in Subsection 3.4.

3.1 The SCAN Algorithm

The procedure SCAN performs the parallel prefix computation for any associative operator. Specifically, the prefix problem is to compute the values of X_0, $X_0 \circ X_1$, $X_0 \circ X_1 \circ X_2$, ... , $X_0 \circ X_1 \circ ... \circ X_{n-1}$, given the values X_0, X_1, ... , X_{n-1}. The operator $\circ$ is any arbitrary associative binary operation such as scalar addition (+), vector addition (+), the minimum operator (MIN) 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 [7], [8], [9], and various VLSI circuit implementations have been proposed [9], [10], [12], . 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.

The algorithm for SCAN is best explained first in a recursive way. Denote by $X_{i:j}$ the value of $X_i \circ X_{i+1} \circ \ ... \ \circ X_j$. 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 step, assume SCAN has been performed by the first half-interval of processors pe_0, pe_1, ... , $pe_{\frac{n}{2}-1}$ on the data X_0, X_1, ... , $X_{\frac{n}{2}-1}$, and also by the second half-interval of processors $pe_{\frac{n}{2}}, pe_{\frac{n}{2}+1}$, ... , pe_{n-1} on the data $X_{\frac{n}{2}}$, $X_{\frac{n}{2}+1}$, ... , X_{n-1}. Assume also that the effect of this two SCAN's is that for every $i = 0, 1, ..., \frac{n}{2} - 1$, pe_i has the value $X_{0:i}$ and

the value $X_{0:\frac{n}{2}-1}$, and that for every $i = \frac{n}{2}, \frac{n}{2}+1, ..., n-1$, pe_i has the value $X_{\frac{n}{2}:i}$ and the value $X_{\frac{n}{2}:n-1}$. After the recursive step, every processor pe_i, for $i = 0, 1, ..., n-1$, 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-interval sends the value $X_{0:\frac{n}{2}-1}$ to $pe_{i+\frac{n}{2}}$ in the second half-interval. Similarly, every processor $pe_{i+\frac{n}{2}}$ in the second half-interval sends the value $X_{\frac{n}{2}:n-1}$ to pe_i in the first half-interval. Afterwards, every pe_i in the second half-interval computes $X_{0:i} := X_{0:\frac{n}{2}-1} \circ X_{\frac{n}{2}:i}$. Finally, every pe_i in both half-intervals computes $X_{0:n-1} := X_{0:\frac{n}{2}-1} \circ X_{\frac{n}{2}:n-1}$. By this recursive 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 $\frac{n}{2^i}$ independent intervals $(I_j)_{0 \le j \le \frac{n}{2^i}-1}$, where $I_j = [j2^i, (j+1)2^i - 1] = \{j2^i, j2^i + 1, j2^i + 2, \ldots, (j+1)2^i - 1\}$. Each interval I_j is in turn divided into two half-intervals 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-interval has the values $X_{j2^i:l}$ and $X_{j2^i:j2^i+2^{i-1}-1}$, and every pe_l in the second half-interval 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 half-intervals perform the same job on their data as the two half-intervals in the recursive 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 impelements 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:

for $j = m$ **to** k **pardo**

    $proc_j$;

endfor

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

The second is of the form: pe_l **does**: S; which means that processors 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 interval I_j = [j2^i, (j+1)2^i - 1] of pe's*/
        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} o 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} o X_{j2^i+2^(i-1):(j+1)2^i-1};
        endfor
    endfor
end
```

The Algorithm for the i-th Stage of SCAN

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

```
Algorithm SCAN(X(0..n - 1));
begin
    for i = 1 to log n do
        Stage(i);
    endfor
end
```

The Algorithm for SCAN

Communication and Complexity analysis of SCAN

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 $i = 1, 2, ..., \log n$ and $j2^i \le l \le j2^i + 2^{i-1} - 1$. When l is expressed as a binary number $l_{n-1}...l_1 l_0$, 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 SCAN 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 SCAN. As for the time complexity,

the body of the two inner for-loops in **Stage** takes constant time. Since all the for-loops in **Stage** are parallel loops, **Stage**(i) takes constant time. It follows that SCAN takes $O(\log n)$ time. Finally, as explained earlier, every processor needs to store two values only at any stage. Thus the space complexity is optimal.

Note that when the operator $\circ$ is scalar addition, we refer to SCAN as ADD-SCAN, when $\circ$ is vector addition, SCAN is referred to as VADD-SCAN, and when $\circ$ is the minimum operator MIN, we refer to SCAN as MIN-SCAN.

3.2 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}, ... , $T_{A_{n-1}}$, the points P_0, P_1, ... , P_{n-1} of the random walk are found by calling VADD-SCAN($A(0..n-1)$), where $A(0) = P_0$ and $A(i) = A_i$ for $i = 1, 2, ..., n-1$. Since $P_i = P_0 + A_1 + A_2 + ... + A_i$, 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 compute a certain flag f_i as follows:

Check-Boundary(P_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, ... , f_{n-1}. This can be accomplished by executing MIN-SCAN($f(0..n-1)$). The minimum min is $f_{0:n-1}$ which is available at every pe_i at the end of MIN-SCAN. 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; in either case $f_l = l < f_i$. 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, and if there is a boundary point, min is equal to the index of the first boundary point (which is $< 2n$), 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_j checks if $min = 2n$ (recall that every pe_j has the min after MIN-SCAN). If pe_j finds $min = 2n$ or $j \leq min$, then pe_j computes the value $x_j = \frac{F(P_j)}{r(P_j)} h^2$ because the point P_j is on the walk. Otherwise, the point P_j is after the first boundary point, in which case pe_j sets x_j to 0.

Now if $min \neq 2n$, then P_{min} is the first boudary 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, ... , pe_{n-1} sum their x_i's to form $Z(W)$. This is accomplished by executing ADD-SCAN($x(0..n-1)$).

On the other hand, if $min = 2n$, then no boundary point has been reached yet. The sum $x_0 + x_1 + ... + x_{n-1}$ is computed using ADD-SCAN($x(0..n-1)$), and the~ 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 variable, 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 in pe_0. The process is repeated till a boundary point is reached.

The execution time of the overall algorithm for parallel RWC is $O(\frac{L}{n} \log n)$ on a hypercube architecture of n processors, where L is the length of the random walk. This is so because each iteration of the algorithm corresponding to a portion of $n-1$ points of the path take $O(\log n)$ time.

This brings an end to the parallel RWC algorithm for rectangular grids and elliptic PDE's. In the next two subsections, non-rectangular regions are handled and then the modifications needed to handle the parabolic PDE's are presented.

3.3 Handling Non-Rectangular Regions

For the case of non-rectangular regions, the region is embedded in the smallest rectangular $m \times k$ grid where the boundary lines of the grid are tangent to the region. 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 is the same as in the rectangular case except for boundary checking. A point $P = (p, q)$ is a boundary point if $p = 0$ or $S(q)$, $p = m - 1$ or $N(q)$, $q = 0$ or $W(p)$, or $q = k - 1$ or $E(p)$. Thus the Check-Boundary procedure becomes:

Check-Boundary(P_i) /* done by pe_i, $P_i = (a_i, b_i)$*/

begin

if ($a_i = 0$ **or** $a_i = S(b_i)$ **or** $a_i = m - 1$ **or** $a_i = N(b_i)$)

or ($b_i = 0$ **or** $b_i = W(a_i)$ **or** $b_i = k - 1$ **or** $b_i = E(a_i)$) **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 simplicty and speed, requiring only some additional storage for the intersection points between the region boundary and the grid.

3.4 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 boundary points and the slight modification of $Z(W)$.

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 $x_j := \frac{F_{T-j}(P_j)}{r_{T-j}(P_j)} h^2$.

and then pe_{min} sets x_{min} to $x_{min} + \phi_{T-min}(P_{min})$ as required to compute $Z(W)$ of equation (7). All the remaining pe's set their x's to 0. $Z(W)$ is then computed by executing ADD-SCAN as in the elliptic case.

If on the other hand, $T - min < 0$ (i.e., run out of time), then the absorbing point is P_T. In this case, only the pe_j's where $j \leq T$ compute $x_j := \frac{F_{T-j}(P_j)}{r_{T-j}(P_j)} h^2$, while all the remaining pe's set their x's to 0. The remaining part to compute $Z(W)$ is the same as in the previous paragraph.

If $min = 2n$ and 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 updated: $T := T - (n-1)$. Afterwards, the algorithm is repeated till an absorbing point is reached.

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

§4. Conclusions

We have presented in this paper a parallel algorithm for the construction of random walks in 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. In the latter case, a certain amount of space is needed at every processor, and it is open whether it can be done in constant space per processor.

This parallel generation 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.

Finally, the parallel generation algorithm presented here is easily generalizable to grids of any dimensions and can have applications in other areas. Future work will investigate parallel random walk generation in grids of geometries different from rectangular grids.

§5. References

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

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

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

[4] J. H. Curtiss, "Sampling methods applied to diffrential and diffrence equations," *Proc. Seminar Scientific Computations*, IBM 1949.

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

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

[7] 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.

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

[9] R. E. Ladner and M. J. Fischer, "Parallel prefix Computation," *Journal of ACM* Vol. 27, pp. 831–838, 1980.

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

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

[12] E. Sadeh, "A Monte Carlo computer for the solution of partial differential equations," M.S. thesis, Washington University, St. Louis, Mo.

[13] M. Snir, "Depth-size trade-off for parallel prefix computation," *Journal of Algorithms*, Vol. 7, pp. 185–201, 1986.