MORE PARALLELISM INTO THE MONTE CARLO SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS

Bruce W. Arden
Addou S. Youssef

CS-TR-061-86

November 1986

More Parallelism into the Monte Carlo Solution of Partial Differential Equations

BRUCE W. ARDEN

College of Engeneering and Applied Science
University of Rochester
Rochester, N.Y. 14627

ABDOU S. YOUSSEF

Departement of Computer Science
Princeton University
Princeton, N.J. 08544

ABSTRACT

The Monte Carlo Method has been studied and used to solve elliptic and parabolic partial differential equations. It has several numerical and computational advantages over other methods. The main computational advantage is the great amount of inherent parallelism it manifests. However, an often costly part of the method has remained sequential. It is the random-walk computation (RWC).

In this report, we parallelize (RWC) using fan-in and fan-out methods. The parallel algorithm takes O(log n) time while the sequential one takes O(n) where n is the average random-walk length.

I . INTRODUCTION:

The Monte Carlo Method has been studied and used to solve elliptic and parabolic partial differential equations [5] – [7]. 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 great parallelism .

The great amount of inherent parallelism is drawn from the fact that the solution at different points are independent, paving the way to independent processes that can run in parallel. Moreover, the solution at each point consists of evaluating a "primary estimator" along a big number of random walks, then averaging these values. The random walks are independent, so here too the estimations along the random walks can be computed in parallel.

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 parallelize the construction of random walks and along with it the evaluation of the primary estimator (this is called intra-walk parallelism), reducing the time of this part of the solution from $O(n)$ to $O(\log n)$ where n is the length of the random walk.

In section II, the Monte Carlo Method for PDE's is presented briefly, and all its possible areas of parallelism are pointed out. Section III, the main section of the paper, introduces the intra-walk parallelism and presents the parallel algorithm for the random walk construction. In section IV we analyze the complexity of the algorithm and we conclude with a few remarks in section V .

II. THE MONTE CARLO METHOD AND ITS INHERENT PARALLELISM:

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

be a PDE and D a region with boundary C. A, B, C, D, E and F 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 D.

Solve equation (1) subject to the boundary condition:

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

B. The parabolic PDE problem:

U, A, B, C, D, E and F are time-dependent and $B^2 - AC = 0$ on D.

Solve equation (1) subject to:

Boundary condition: $U(x,y,t) = \phi(x,y,t)$ if $(x,y) \in C$ (3)

Initial condition: $U(x,y,0) = g(x,y)$ if $(x,y) \in D$ (4)

The region D is divided into a regular grid of size h. Each point P of the grid (except the boundary points) has five neighbors P_1, P_2, P_3, P_4, P_5 as depicted in fig.1. We denote by d_i the direction along which we move from P to P_i where $i = 1, 2, 3, 4, 5$. A random number generator (RNG) generates random directions (i.e., d_1, d_2, d_3, d_4, d_5). A random walk starting at P is constructed by moving away from P following directions generated by (RNG) till an absorbing point is hit. In the elliptic case, the absorbing points are the boundary points, while in the parabolic case, they are either boundary points or points reached at time point 0.

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 P.

Let W_i be a random walk starting at P and ending at a boundary point Q_i, and

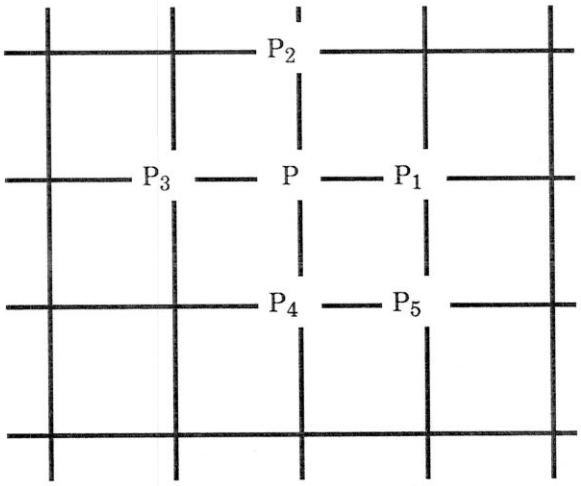


Fig.1

$$Z(W_i) = V(Q_i) + h^2 \sum_{P_j \in W_i} \frac{F(P_j)}{r(P_j)} \qquad (5)$$

The Monte Carlo solution of the elliptic equation (1) at point P, subject to (2), consists of generating a number of random walks W_1, W_2, ..., W_N, all staring at P and ending at Q_1, Q_2, ..., Q_N, respectively. Then, $Z(W_i)$ is evaluated (or $Z(W_i)$ is evaluated while generating W_i). Afterwards, $U(P)$ 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 $U(P)$, and θ the secondary estimator. For the proof that this method yields a good approximation of U, see [4], [8].

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., $t_n = nk$, $n \geq 0$), and $U(P)$, A, B, C, D, E, F and $r(P)$ at time t_n are denoted $U_n(P)$, A_n, B_n, C_n, D_n, E_n, F_n and $r_n(P)$, respectively.

Random walks W's are constructed as before except that W is started at P at time $t_n = nk$, and at each step (following a new direction), the time is decreased one unit.

W is finished if either a boundary point is reached or time runs out (after n steps), whichever comes first.

In this case,

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

$$V_s(Q_i) = \begin{cases} \phi_s(Q_i) & \text{if } Q_i \in C,\ s \geq 0 \\ g(Q_i) & \text{if } s = 0 \end{cases}$$

(i.e., Q_i is reached at time 0, and may be a non-boundary point)

The Monte Carlo solution of the parabolic equation (1) at point P, at time t_n, subject to (3) and (4) consists of generating W_1, W_2, ..., W_N, evaluating the $Z(W_i)$'s and averaging them, as in the previous case.

Now it can be clearly seen that the random walks W_1, W_2, ..., W_N are independent, $Z(W_1)$, $Z(W_2)$, ..., $Z(W_N)$ can be computed independently (and thus in parallel). This inter-walk parallelism has been studied in [2], [3], [7]. It is also clear to see that U can be computed at different points independently (and thus in parallel).

The third candidate for parallelism is the generation of a random walk W_i, and the computation of $Z(W_i)$ along with it. We call the whole thing random walk computation (RWC).

We shall describe the sequential (RWC) next. The computation of $Z(W_i)$ in the elliptic case is carried out as follows:

```
x = 0;
temp-end = P;      {holds the temporary end-point of the walk}
is-boundary = false;

while(is-boundary = false) do
   begin
      is-boundary = chech-boundary(temp-end);
```

if (is-boundary = **false**)
$x = x + F$(temp-end) / r(temp-end);
d = RNG(); {a random direction}
temp-end = new-node(temp-end,d); { updates the temporary end-point using the previous end and the direction d}
end
$Z = \phi$(temp-end) + $h*h*x$;

Algorithm 1.

The parabolic case is quite similar and will not be treated individually.

Some slight parallelism is obvious: the RNG, the updating of end-point, and the checking of boundary-crossing are independent and can run in parallel. This parallelism along with the inter-walk parallelism has been studied in [7]. Different implementation schemes on different machine architectures such as SIMD, MIMD, etc. ... have been studied in [1], [2], [3], [7].

The balk of the computation time remains in the sequentiality of the random walk generation (the **while**-loop runs as many times as the random walk length). At first glance, this sequentiality seems inherent. However, it can be parallelized, cutting down the RWC time from O(n) to O(log$n$) where n is the random walk length. This is the subject of the next sections.

III. NEW INTRA-WALK PARALLELISM:

The first idea is to have a number of independent RNG's rather than only one. They can be thought of as a multiple random number generator (MRNG) that generates sequences of random numbers. Assume there are n RNG's and $n = 2^k$.

The problem can now be cast as follows: Given a grid, a point P of the grid, and a sequence of random directions d_{i_1}, d_{i_2}, ..., d_{i_n} (taken from the five directions introduced earlier), construct in parallel the random walk that starts at P and moves away following direction $d_{i_1}, d_{i_2}, ..., d_{i_n}$, consecutively.

Suppose that the grid points are numbered from 1 to S, row-wise. We can think of $d_{i_1}, d_{i_2}, ..., d_{i_n}$ as functions acting on integers. As an example, take the grid in fig.2.

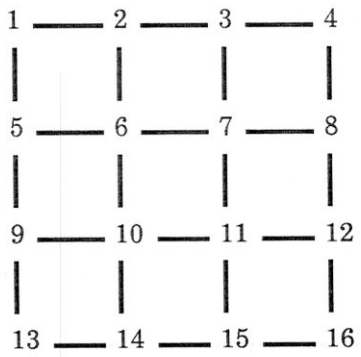


Fig.2.

$d_{i_1} = i + 1; d_{i_2} = i - 4; d_{i_3} = i - 1; d_{i_1} = i + 4; d_{i_1} = i + 5.$
To simplify the notation, let $f_1 = d_{i_1}, ..., f_n = d_{i_n}$.
Let $P_1 = f_1(P)$, $P_2 = f_2(P_1)$, ..., $P_n = f_n(P_{n-1})$. P, P_1, P_2, ..., P_n is the random walk sought. the problem then is to find $P_1, P_2, ..., P_n$ in parallel, and compute $Z(W)$ as the search for $P_1, P_2, ..., P_n$ takes place.

The algorithm consists of three phases. Phase I involves the following computation:

$$Step\ 1: f_1^{(1)} = f_2 f_1, f_2^{(1)} = f_4 f_3, ..., f_{n/2}^{(1)} = f_n f_{n-1}$$

$$Step\ 2: f_1^{(2)} = f_2^{(1)} f_1^{(1)}, f_2^{(2)} = f_4^{(1)} f_3^{(1)}, ..., f_{n/4}^{(2)} = f_{n/2}^{(1)} f_{n/2-1}^{(1)}$$

$$Step\ i: f_1^{(i)} = f_2^{(i-1)} f_1^{(i-1)}, f_2^{(i)} = f_4^{(i-1)} f_3^{(i-1)}, ..., f_{n/2^i}^{(i)} = f_{n/2^i-1}^{(i-1)} f_{n/2^i-1}^{(i-1)}{}_{-1}$$

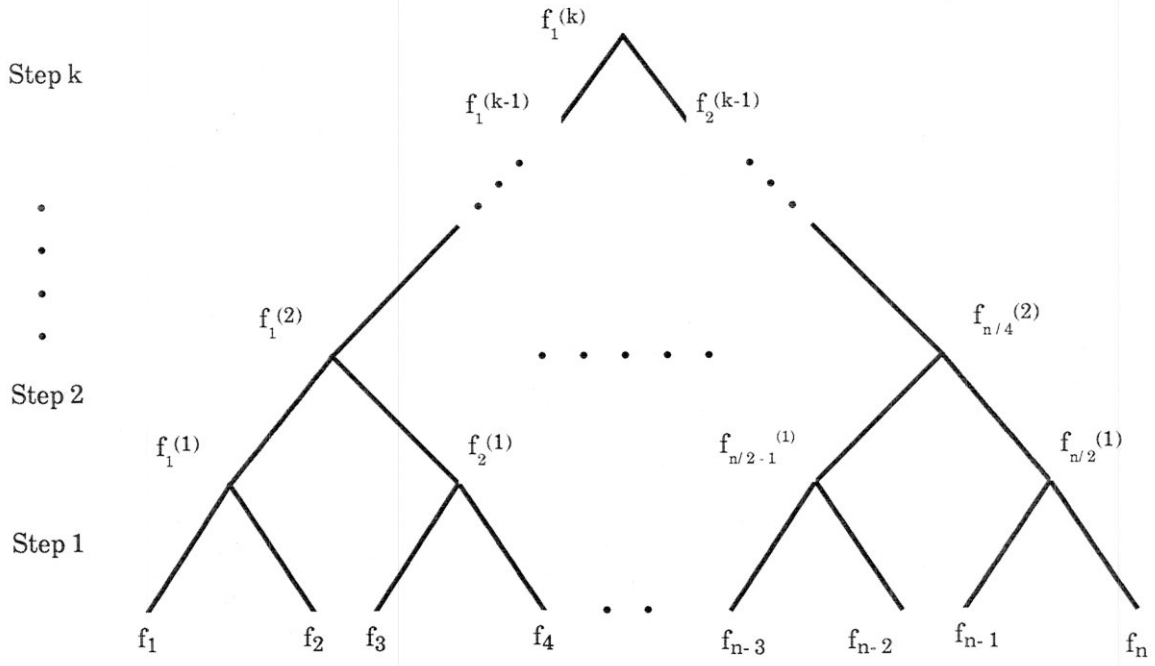
$$Step\ k: f_1^{(k)} = f_2^{(k-1)} f_1^{(k-1)}$$

as depicted in fig.3, in a bottom-up fashion, assuming that $n = 2^k$ for simplicity.

Note that $f_1^{(k)} = f_n f_{n-1} \cdots f_2 f_1$, $f_1^{(k-1)} = f_{n/2} f_{n/2-1} \cdots f_1$, $f_2^{(k-1)} = f_n f_{n-1} \cdots f_{n/2+1}$, $f_1^{(i)} = f_{n/2^i} \cdots f_2 f_1$,

and in general $f_j^{(i)} = f_{2^{i+j-1}} \cdots f_{2^{i+j-2}+2} f_{2^{i+j-2}+1}$, a composition of $n/2^i$ functions.

Each step can be processed in parallel by a number of processing elements (pe's).

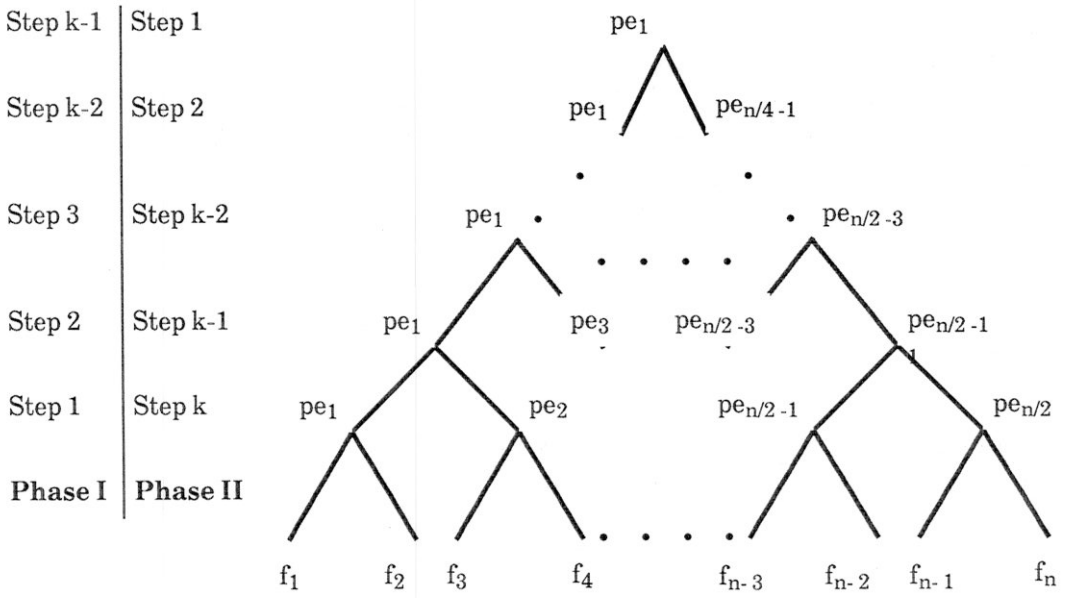


Phase I

Fig.3.

In phase II, P_1, P_2, ..., P_{n-1} are found in a top-down fashion as depicted in fig.5 where the indices of the points determined at the nodes in fig.5 coincide exactly with the numbers of the nodes of that tree if it is a binary search tree with keys 1, 2, ..., n-1. At the end, P_n is determined by pe_1. Note that the assignment of the pe's is the same as in fig.4. Along with the determination of P_1, P_2, ..., P_n goes the boundary-crossing checking and the necessary measures to be taken in this case. This will be explained in more detail later.

In phase III, the terms of the summation part of (5) (or (7)) that correspond to the points found and proved to be within boundary in phase II are summed in a bottom-up fashion, with the same assignment of pe's as in fig.4.



Assignment of pe's for phase I and II

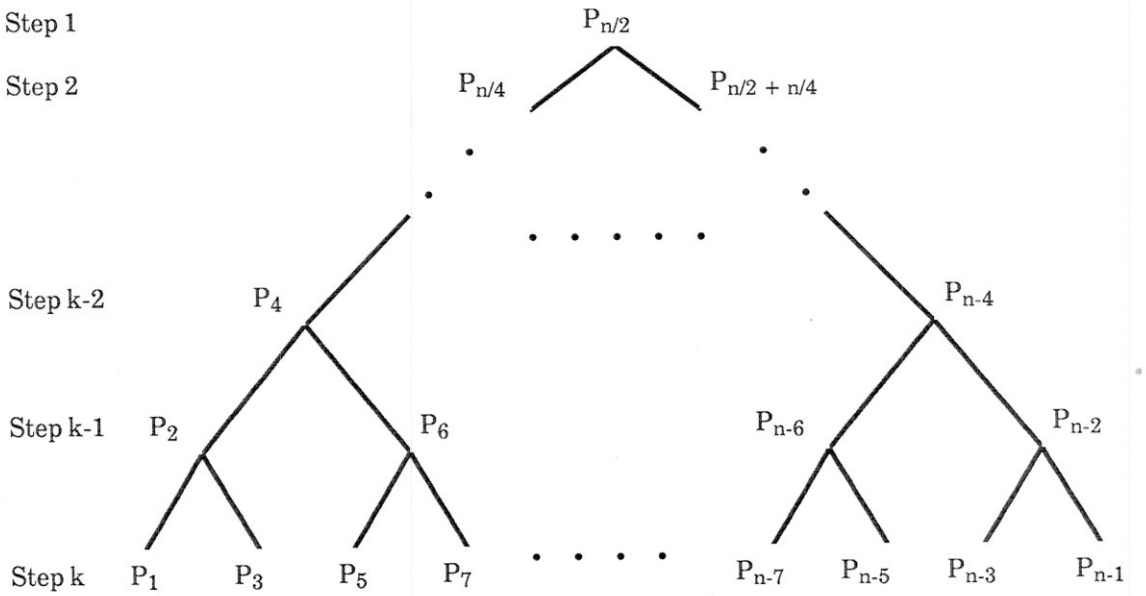
Fig.4.

If the boundary is not reached yet, a new sequence of random directions is generated and the three phases are repeated with this new sequence and the last point reached (in the previous iteration) as input. This cycling continues until an absorbing point Q is reached. During the cycling, the partial sums computed in phase III are accumulated. In the final step, Z is computed by multiplying the accumulated sum by h^2 and then adding it to $\phi(Q)$.

Several questions arise at this point:

(i) Whether the composition of those functions can be carried out easily and independently from arguments. The answer is positive, as will be seen later, making phase I implementable.

(ii) How to compose those functions (in order to find the points in phase II).



Phase II

Fig.5.

(iii) Whether the resulting functions reveal if there is any boundary crossing and where. The answer is negative.

(iv) What measures to take to detect boundary crossing (in order to make phase II implementable).

We handle these questions next.

If the region is a full grid as in fig.6, we number its nodes row-wise from 1 to S. However, if it is irregular as in fig.7, we embed it in a full grid, but we delete two exterior nodes from each row and each column so that all the rows have the same number of nodes and so do all the columns (fig.8), then we number it row-wise.. This numbering scheme is used to make the five direction-functions simple. Suppose the row size is m. Then, $d_1(i) = i + 1$, $d_2(i) = i - m$, $d_3(i) = i - 1$, $d_4(i) = i + m$, and $d_5(i) = i + m + 1$.

The f_i's are taken from $\{d_1, d_2, d_3, d_4, d_5\}$, and their composition is the composition of the d_i's. Note that the d_i's are "translations" of the form t_a where $t_a(x) = x + a$.

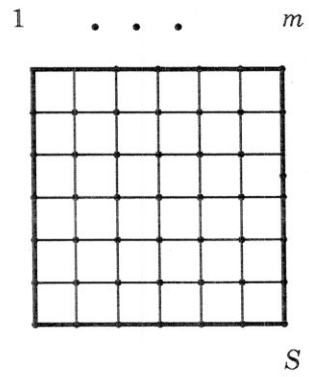


Fig.6.

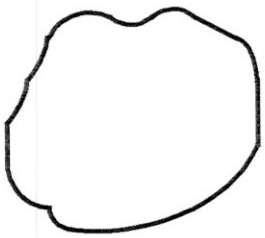


Fig.7.

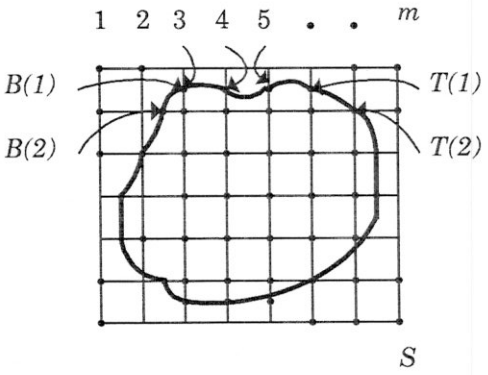


Fig.8.

Since $t_a t_b = t_{a+b}$, each translation t_a can be identified by its parameter a (i.e., its computer representation is the number a), and the composition is done by adding the corresponding parameters without referring to the argument x. This answers question (i). Hence, the functions of phase I are computed by mere additions of their representations, and are themselves translations, answering question (ii). It is important to note the distinction between *computation* of a function and *evaluation* of a function at some integer; the first yields a function while the second yields an integer.

As to question (iii), note that since $f_j^{(i)}$ is a translation (of parameter a_{ij}, say), a_{ij} does not provide any information as to whether the boundary is crossed (except in the case where $a_{ij} > S$).

Now we come to the last and most important question, the boundary-crossing detection. This will be done in phase II of the algorithm as follows: As the finding of the P_i's proceeds in the top-down fashion, each time a P_i is computed at some pe_j, P_i is checked if it is on boundary, off boundary or within boundary. If P_i is within boundary, the computation proceeds normally to find the other points; if it is off boundary, P_i and all the the P_s for $s>i$ are discarded from the walk (i.e., all the points computed (or will be computed) at the right side of P_i in the tree in fig.5); if it

is on boundary, all P_s for s>i are discarded. Discarding those points can be carried out on a SIMD machine by having the pe that computes P_i report to the control unit that P_i is off boundary; then the control sends instructions to all the pe's responsible for P_s , s>i, to discard those P_s's. In phase III, those discarded points do not contribute to the sum.

The implementation of the checking is discussed next. In case the region is a full grid as in fig.6., it is simple:

Let $I = P_i \bmod m$ (recall that P_i is an integer)
If $P_i > S$ or $P_i < 1$, it is off boundary
If $I = 0$ or 1, it is on boundary
Otherwise, it is within boundary

In case the region is irregular as in fig.7, some data structure has to be kept in the memory of each processing element. After embedding the region in a grid as in fig.8, for each row *i*, we keep *B(i)* and *T(i)*, beginning and ending nodes of row i on the region, respectively.

Let $I = \lceil P_i/m \rceil$ (the row number of P_i)
If $P_i > S$ or $P_i < 1$ or $P_i < B(I)$ or $P_i > T(I)$, it is off boundary
If $P_i = B(I)$ or $P_i = T(I)$, it is on boundary
If $B(I) < P_i < T(I)$, it is within boundary

The full algorithm is shown below.

```
procedure RWC(P)
begin
   x=0;      {holds the accumulated sum}
   P0 = P;   {P0 holds the temporary end of the random walk which is initially the
point P only}
   Z = 0;        {the primary estimator}
   boundary = false;    { a boolean variable to tell if the boundary has been
crossed}
   while(boundary = false) do
      begin
```

```
for i = 1 to n step 1 in parallel do
    begin
        f_i = RNG_i();   {parallel generation of n random directions. f_i = 1,-1,m,-m or m+1}
    end
{phase I next}
for i = 1 to k step 1 do        {i is the step number of fig.3}
    for j = 1 to n/2^i step 1 in parallel do
        f_j^(i) = f_2*j^(i-1) + f_2*j-1^(i-1) ;  { as seen earlier, function composition is equivalent to}
                                {the addition of their representations
}
{phase II next}
for i = k to 1 step = -1 do        {i is the step number in fig.5}
    begin
        P_n = P_0;  { this is for index handling only}
        for j = 1 to n/2^i step 1 in parallel do
            begin
                index = (2*j-1)*2^i-1;   {the index of the j-th point (from left) in the i-th
                                                                    step of fig.5}
                father_index = (⌈(j-1)/2⌉ +2)*2^i-1;   {the index of the father of P_index }
                P_index = f_j^(i) + P_father-index ;      {f(P) = f +P if f is a translation}
                boundary = check_boundary(P_index);
            end

    end
    if(boundary = false) then
        begin
            P_n = f_n + P_n-1;
            boundary = check_boundary(P_n);
            if(boundary = false) then      {updates the temporary end-point of the walk
```

```
                P0 = Pn;                          if the boundary is not yet reached}
            end
        {phase III next}
            for j = 1 to n step 1 in parallel do
                if (Pj is discarded) then
                    xj(0) = 0;
                else  xj(0) = F(Pj)/r(Pj);
            for i = 1 to k step 1 do
                for j = 1 to n/2i step 1 in parallel do
                    xj(i) = x2*j(i-1) + x2*j-1(i-1) ;
        { end of phase III}
            x = x + x1(k);          {update the accumulated sum}
        end {of the while loop)
        Z = h*h*x + φ(P0);          {final value of the random walk}
end
```

Algorithm 2.

The check_boundary procedure is presented next for the general (i.e., irregular) region :

```
procedure check_boundary(Pi)
begin
   I = ⌈Pi /m⌉ ;
   if(Pi>S or Pi<1 or Pi<B(I) or Pi>T(I)) then
      begin
         discard all Ps for s ≥ i;      {done by the control unit}
         boundary = true;
      end
   if(Pi = B(I) or Pi = T(I)) then
      begin
         discard all Ps for s > i;      {done by the control unit}
         boundary = true;
         P0 = Pi;       {update endpoint of the walk}
      end
   return(boundary);
```

end

Algorithm 2 is for elliptic equations. For parabolic equations, it needs only minor, straightforward modifications.

IV. COMPLEXITY OF THE ALGORITHM:

This section discusses the computation time, the number of pe's needed for full parallelism, and the communication time.

Let n be the average random-walk length, p the number of pe's used for the RWC, and q the number of RNG's.

The sequential time of RWC is $O(n)$ because the while-loop of algorithm 1 loops n times. For the parallel time (of algorithm 2), suppose first that $p \geq n/2$. The expected number of times algorithm 2 will loop is $\lceil q/n \rceil$. Each step of phase I takes one time unit because it involves one addition (actually many additions all done in parallel because the available pe's are more than the addition operations). Thus, phase I takes $k = \log n$ time.Each of the remaining two phases have similar computational patterns (i.e., k steps, each executes in parallel and takes a constant time). Therefore, the three phases take $O(\log n)$ time, and the whole algorithm takes $O(\lceil q/n \rceil \log n)$ time $= O(\log n)$ if $q = n$.

If $p \leq n/2$, each of the three phases takes $O(\lceil (n/2)/p \rceil \log n)$ time; consequently, the whole algorithm takes $O(\lceil q/n \rceil \lceil (n/2)/p \rceil \log n)$ time.

To complete the solution at one point, the $Z(W_i)$'s (N of them) have to be computed and averaged. Their computation can be done in parallel taking Np pe's and $O(\lceil q/n \rceil \lceil (n/2)/p \rceil \log n)$ time. Averaging them takes $\log N + 1$ time units on N pe's.

As a result, $U(P)$ takes $O(\log n) + \log N + 1$ time on $Nn/2$ pe's if the RWC is fully parallelized, and $O(n) + \log N + 1$ time if RWC is run sequentially.

The above figures are for the computation time. For the communication time, fig.3, fig.4 and fig.5 show that the three phases have the same communication patterns: A binary tree pattern, bottom-up or top-down. the communication

implementation is architecture-dependent.Different architectures are considered below, yielding different communication times for algorithm 2. The proof should be straightforward.

(i) Mesh-connected machine of size p (perfect square): $2(\sqrt{p}-1)$ communication steps.

(ii) Hypercubes of size p (power of 2): log$p$ steps.

(iii) Omega- or Benes-connected machines of size p: log$p$ permutations.

Same argument applies to the computation of θ of (6) because it has the same communication pattern.

To keep the communication time at the same complexity as that of the computation time (i.e., $O(\log N)+O(\log n)$) time, it is recommended to use hypercubes, omega-connected machines or Benes-connected machines.

From the analysis above, it is concluded that if the grid is large and fine-grained, then there are very many grid points and n is expected to be very large, and parallel RWC offers great speed-up (in the order of n/logn). On the other hand, if n is relatively small, the speed-up is small and may not warrant the cost of extra processing elements.

It should be noted that if the number of available pe's is less than or equal the number of random walks, then running RWC in parallel would increase the overall computation time of θ. *Consequently, Parallel RWC should be used only when the number of pe's exceeds that of the random walks.*

V. CONCLUSION:

We have parallelized the part that was the most sequential and often costly of the whole Monte Carlo solution of partial differential equations, reducing the time of this solution to what we conjecture to be the minimum. In parallelizing RWC, we used a principle that underlies many parallel algorithms. This principle involves the formulation of a problem as a composition of mathematical functions which can be composed rapidly and independently of arguments (or input). Then these functions are composed in a binary-tree fashion (as in fig.3), similar to any semi-group computation (e.g., addition or multiplication of n numbers).

REFERENCES:

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

[2] V. C. Bhavsar and V. V. Kanetkar, "A multiple microprocessor system (MMPS) for the MonteCarlo 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 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 differential and difference equations," *Proc. Seminar Scientific Computation*, I.B.M., 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] E. Sadeh and M. A. Franklin, " Monte Carlo solutions of partial differential equations by special purpose digital computer," I*EEE Tràns. Comput.*, C–23, pp. 389–397, Apr. 1974.

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