Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms

Richard Cole‡ and Uzi Vishkin†

New York University‡ and Tel Aviv University†

2. The deterministic coin tossing technique

2.1. The basic technique

We illustrate the deterministic coin tossing technique by using it to break the (apparently) symmetric situation that arises in the following problem.

Input: A connected directed graph $G(V,E)$. The in-degree of each vertex is exactly one. The out-degree of each vertex is exactly one. Such a graph is called a ring since it forms a directed circuit. Let $n = |V|$.

We define a subset $U$ of $V$ to be an $r$-ruling set of $G$ if:

(1) No two vertices of $U$ are adjacent.

(2) For each vertex $v$ in $V$ there is a directed path from $v$ to some vertex in $U$ whose edge length is at most $r$.

The $r$-ruling set problem: Find an $r$-ruling set of $V$.

In order to demonstrate our basic technique we give an $O(1)$ time algorithm using $n$ processors for the $\lfloor \log n \rfloor$-ruling set problem. The algorithm is given for the EREW PRAM. In section 2.2 we present a recursive application of the technique. It leads to an $O(k)$ time algorithm using $n$ processors for the $\lfloor \log^k n \rfloor$-ruling set problem. In particular, it provides an $O(\log n)$ time algorithm using $n$ processors for the 2-ruling set problem. In section 2.3 we describe a non-recursive approach that provides an $O(\log n)$ time algorithm using $n/\log n$ processors for the 2-ruling set problem.
Assumptions about the input representation: The vertices are given in an array of length $n$. The entries of the array are numbered from $0$ to $n-1$. The numbers are represented as binary strings of length $\lceil \log n \rceil$. We refer to each binary symbol (bit) of this representation by a number between $0$ and $\lceil \log n \rceil - 1$. The rightmost (least significant) bit is called bit number $0$ and the leftmost bit is called bit number $\lceil \log n \rceil - 1$. Each vertex has a pointer to the next vertex in the ring (representing its outgoing edge). For simplicity we assume that $\log n$ is an integer**.

Here is a verbal description of an algorithm for the log $n$-ruling set problem. The algorithm is given later. Processor $i$, $0 \leq i \leq n-1$, is assigned to entry $i$ of the input array (for simplicity, entry $i$ is called vertex $i$). It will attach the number $i$ to vertex $i$. So, the present "serial" number of vertex $i$, denoted $\text{SERIAL}_0(i)$, is $i$. Next, we attach to vertex $i$ a new serial number, denoted $\text{SERIAL}_1(i)$, as follows. Let $i_2$ be the vertex following $i$. (That is $(i, i_2)$ is in $E$). Let $j$ be "the index of the rightmost bit in which $i$ and $i_2$ differ". Processor $i$ assigns $j$ to $\text{SERIAL}_1(i)$.

** Example. Let $i$ be $\ldots 010101$ and $i_2$ be $\ldots 111101$. The index of the rightmost bit in which $i$ and $i_2$ differ is $3$ (recall the rightmost bit has number $0$). Therefore, $\text{SERIAL}_1(i)$ is $3$. **
Remark (Due to B. Schieber). \( j \) can be computed by a constant number of standard operations, as follows. Without loss of generality suppose \( i \geq i_3 \) (otherwise interchange the two numbers). Set \( h = i - i_3 \), and \( k = h - 1 \). (So \( h \) has a 1 for bit number \( j \), and a 0 for bits of lesser significance, while \( k \) has a 0 for bit number \( j \), and a 1 for bits of lesser significance; also, \( h \) and \( k \) agree on the bits of higher significance.) Compute \( l = h \oplus k \), where \( \oplus \) is the exclusive-or operation. We observe \( l \) is the unary representation of \( j + 1 \). So it just remains to convert this value from unary to binary, and then to subtract one.

Next, we show how to use the information in vector \( SERIAL_1 \) in order to find a \( \log n \)-ruling set.

Fact 1: For all \( i \), \( SERIAL_1(i) \) is a number between 0 and \( \log n - 1 \) and needs only \( \lceil \log \log n \rceil \) bits for its representation. For simplicity we will assume that \( \log \log n \) is an integer.

Let \( i_1 \) and \( i_2 \) be, respectively, the vertices preceding and following \( i \). \( SERIAL_1(i) \) is a local minimum if \( SERIAL_1(i) \leq SERIAL_1(i_1) \) and \( SERIAL_1(i) \leq SERIAL_1(i_2) \). A local maximum is defined similarly.

Fact 2: The number of vertices in the shortest path from any vertex in \( G \) to the next (vertex that provides a) local extremum (maximum or minimum), with respect to \( SERIAL_1 \), is at most \( \log n \).

Observe that several local minima (or maxima) may form a "chain" of successive vertices in \( G \). Requirement (1), in the definition of an \( r \)-ruling set, does not allow us to include all these local minima in the set of selected vertices. Our algorithm exploits the alternation property (defined below) of vector \( SERIAL_1 \) to overcome this problem.
The alternation property: Let \( i \) be a vertex and \( j \) be its successor. If bit number \( \text{SERIAL}_1(i) \) of \( \text{SERIAL}_0(i) \) is 0 (resp. 1), then this bit is 1 (resp. 0) in \( \text{SERIAL}_0(j) \). (For \( \text{SERIAL}_1(i) \) is the index of the rightmost bit on which \( \text{SERIAL}_0(i) \) and \( \text{SERIAL}_0(j) \) differ.)

Suppose that \( i_1, i_2, \ldots \) is a chain in \( G \) such that \( \text{SERIAL}_1(i) \) is a local minimum (resp. maximum) for every \( i \) in the chain. Then:

**Fact 3:** For all vertices in the chain \( \text{SERIAL}_1 \) is the same (i.e., \( \text{SERIAL}_1(i_1) = \text{SERIAL}_1(i_2) = \cdots \)). (By definition of local minimum).

Below, we consider bit number \( \text{SERIAL}_1(i_1) \) of \( \text{SERIAL}_0 \) for all vertices in the chain.

**Fact 4:** The following sequence of bits is an alternating sequence of zeros and ones.

- Bit number \( \text{SERIAL}_1(i_1) \) of \( \text{SERIAL}_0(i_1) \), bit number \( \text{SERIAL}_1(i_2) (=\text{SERIAL}_1(i_1)) \) of \( \text{SERIAL}_0(i_2) \), ..., bit number \( \text{SERIAL}_1(i_j) (=\text{SERIAL}_1(i_i)) \) of \( \text{SERIAL}_0(i_j) \), ...

(This is readily implied by the alternation property.)

We can now understand why we called our technique deterministic coin tossing. We associated zeros and ones with the vertices, based on their original serial numbers; these serial numbers were set deterministically. This association allows us to treat (apparently) similar vertices differently. Finally, note that coin tossing can be used for similar purposes.

We return to the algorithm. We select the following subset of vertices.

We select all vertices \( i \) that are local minima and satisfy one of the following two conditions:

1. Neither of \( i \)'s neighbors (the vertices adjacent to \( i \)) is a local minimum.
2. Bit number \( \text{SERIAL}_1(i) \) is 1.

We say an unselected vertex is available if neither of its neighbors was selected and it is a local maximum. We select all available vertices \( i \) that satisfy one of the following two properties.

1. Neither of \( i \)'s neighbors is available.
2. Bit number \( \text{SERIAL}_1(i) \) is 1.

The selected vertices form a log \( n \)-ruling set. Requirement (1) is satisfied since we never select two adjacent vertices. Requirement (2) is satisfied by Fact 2 and since every local extremum is either selected or is a neighbor of a vertex that was selected.
In computer science, the iterated logarithm of $n$, written $\log^* n$ (usually read "log star"), is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1. The simplest formal definition is the result of this recursive function:

\[
\log^* n := \begin{cases} 
0 & \text{if } n \leq 1; \\
1 + \log^*(\log n) & \text{if } n > 1
\end{cases}
\]
2.2. The k-th application of the basic step.

In order to prepare the input for the k-th application of the basic step, we "delete" from G the vertices that were selected in the previous \( k-1 \) applications, their neighbors, and the edges incident to any vertex being deleted.

The input for the k-th application of the basic step is the remaining graph and vector \( SERIAL_{k-1} \). \( SERIAL_{k-1} \) will play the role played above by \( SERIAL_0 \) and a new vector \( SERIAL_k \) will play the role of \( SERIAL_1 \). The degree of each vertex in the input graph is at most 2 (if the directions of the edges are ignored). It is very simple to extend the basic step to handle vertices whose degree is \( \leq 1 \). Vertices whose degree is 2 are treated as in the basic step (unless they have a neighbor whose degree is 1). The k-th application of the basic step will be as follows. (For an explanation see Fact 5 below.)

for processor \( i, \ 0 \leq i \leq n-1 \), pardo

if vertex \( i \) or one of its neighbors have been selected
in a previous application of the basic step

then "delete" vertex \( i \) and the edges incident to it

for processor \( i, \ 0 \leq i \leq n-1 \), such that \( i \) is in the remaining graph pardo

case 1 \( deg(i) = 2 \)

then compute \( SERIAL_k(i) \)

if the degree of each of \( i \)'s two neighbors is 2

then apply the basic step to \( i \)

case 2 \( deg(i) = 0 \)

then select \( i \)

case 3 \( deg(i) = 1 \)

then if either of the following is satisfied

(1) the degree of \( i \)'s neighbor is 2

(2) \( i \)'s neighbor is its successor

then select \( i \)

The following fact helps to clarify the operation of the k-th application of the basic step.
Fact 5: Let $i,j$ be adjacent in the input graph for the $k$-th application. Then:

\[ SERIAL_{k-1}(i) \neq SERIAL_{k-1}(j). \] (For $k-1$ this inequality clearly holds. We show that it also holds if $k > 1$. If they were equal each of them had to be a local maximum or local minimum at the $(k-1)$-st application. The selection of the ruling set implies that each local maximum or local minimum $v$ is either selected or has a neighbor that is selected. Therefore, $v$ must have been deleted and cannot be included in this input graph).

Fact 6: It is easy to deduce that the output graph consists of simple paths each comprising most $\log \log \ldots \log n$ vertices where the sequence includes $k$ "log"s. (Again, we assume for simplicity that each application of a sequence of logs to $n$ produces only integers).

We finish this description with three obvious conclusions.

1. After a total of $\log n$ applications we delete all vertices in the graph.
2. The vertices that were selected form a 2-ruling set.
3. The cardinality of a 2-ruling set (in a ring) is at least $n/3$.

If our original input is a directed path of $n$ vertices, rather than a ring, we obtain a 2-ruling set by applying the basic step $\log n$ times, as above. To obtain a $\log^k n$-ruling set we apply the basic step $k$ times. We have shown:

Theorem 2.2: A $\log^k n$-ruling set can be obtained in $O(k)$ time using $n$ processors.

Corollary 2.1: A 2-ruling set can be obtained in $O(\log n)$ time using $n$ processors.

General remarks.

1. Readers familiar with randomized algorithms may be tempted to solve these problems using randomization. We already mentioned that [Vi-84b] did so for the (related) list ranking problem. Our deterministic technique was inspired by such a randomized approach.

2. The $[\log n]$-ruling set algorithm is valid even for models of distributed computation that allow only local communication and do not have a shared memory like a PRAM. We do not elaborate on this.
2.3. An optimal 2-ruling set algorithm

First, we find a log_2 n-ruling set using the basic step, above. Below, we describe how to add more vertices to the log_2 n-ruling set to produce a 2-ruling set. These additional vertices are selected using the numbers SERIAL_i associated with each vertex, as follows.

for i = 0 to log_2 n - 1 do
    for each vertex v for which SERIAL_i(v) = i pardo
        if v is not in the ruling set and neither of the neighbors of v is in the ruling set
            then add v to the ruling set

Note that if SERIAL_i(v) = i, and if neither v nor its neighbors are in the ruling set, then neither of the neighbors w of v has SERIAL_i(w) = i. Thus this procedure selects a set of non-adjacent vertices. When the procedure is finished, any vertex that was not selected must have a selected vertex as a neighbor. Thus this procedure selects a 2-ruling set.

Clearly, the procedure can run in O(log n) time. At first sight, it appears to require Θ(n) processors to achieve this running time (simply assign a processor to each vertex v). We show that, in fact, this time can be achieved using only n/log n processors. To do this we perform two instructions:

Instruction 1. We sort the vertices by their SERIAL_i number. The outcome of this sort is that each vertex v will be given a number RANK(v), 1 ≤ RANK(v) ≤ n. No two vertices will have the same RANK.

Instruction 2. For each v, RANK(v) := RANK(v) + i/n/log n, where i = SERIAL_i(v).

We then process the vertices in 2log n rounds. In round j (1 ≤ j ≤ 2log n), we process all vertices v such that (j - 1)n/log n < RANK(v) ≤ jn/log n.

Instruction 2 guarantees that we never simultaneously process two vertices whose SERIAL_i number is different.

Instruction 1 simply needs a bucket sort of n numbers in the range [0, log n - 1]. The rest of this section shows how to perform such a sort in O(log n) time using n/log n processors. We remark that the bucket sort, while not performed in place, nonetheless will require only O(n) space. It may be helpful to read section 3 at this point; it reviews the prefix sum parallel algorithm, used below.
The sort proceeds in three stages. First, we count, for each number \( i \), the number of vertices \( v \) for which \( \text{SERIAL}_1(v) = i \). Second, using a prefix sum sequential algorithm, we count the number of vertices \( v \) for which \( \text{SERIAL}_1(v) < i \). in \( O(\log n) \) time. Third, for each vertex \( v \), we determine a unique value \( \text{RANK}(v) \). No two vertices get the same \( \text{RANK} \).

The first stage proceeds in two substages. First, we divide the vertices into groups of size \( \log n \). For each group, in \( O(\log n) \) time, using one processor per group, we count the number of vertices \( v \) for which \( \text{SERIAL}_1(v) = i \), \( 0 \leq i < \log n \). (We also determine, on the fly, for each vertex \( v \), how many vertices \( w \), preceding \( v \) in the group, satisfy \( \text{SERIAL}_1(w) = \text{SERIAL}_1(v) \).) We obtain \( n/\log n \) sets of \( \log n \) counts, one set per group. Second, using a prefix sum parallel algorithm (or rather, \( \log n \) of them), for each number \( i \), we sum the \( n/\log n \) associated counts (for each \( i \), one count per group). Clearly, this stage, implemented with \( n/\log n \) processors, uses \( O(\log n) \) time.

The second stage is straightforward. In the third stage, for each vertex \( v \), we compute \( \text{RANK}(v) \) using a single processor and \( O(1) \) time, where several processors may read from the same memory location. (It is easy to simulate this computation in \( O(\log n) \) time using \( n/\log n \) processors on an EREW PRAM). \( \text{RANK}(v) \) will be: one, plus the number of vertices \( u \) such that \( \text{SERIAL}_1(u) < \text{SERIAL}_1(v) \) (computed in the second stage), plus the number of vertices \( w \) such that \( \text{SERIAL}_1(w) = \text{SERIAL}_1(v) \) and \( w \) appears before \( v \) in the input array. The last number is obtained by adding the number of such vertices \( w \) that appear in groups prior to the group of \( v \) and the number of such vertices \( w \) that appear prior to \( v \) in its own group. Both numbers were computed in the first stage.

It now follows that the algorithm for bucket sort, with \( \log n \) buckets, uses \( n/\log n \) processors and \( O(\log n) \) time. We conclude

**Theorem 2.3:** A 2-ruling set can be obtained in \( O(\log n) \) time using \( n/\log n \) processors.

**Remark:** It is easy to modify the bucket sort algorithm to sort \( n \) numbers in the range \([0, m - 1] \), \( m \geq \log n \). The algorithm will use \( n/\log n \) processors and \( O(\log n \frac{\log m}{\log \log n}) \) time. (Each number should be represented using digits that can take on \( \log n \) values; proceed as in the standard bucket sort for multi-digit numbers.) Also, for \( m \geq t \geq \log n \), using \( n/t \) processors, we achieve a time of \( O(t \frac{\log m}{\log t}) \) (replace \( \log n \) by \( t \) in the above algorithm).