Efficient Algorithms for Joint Density Approximations of Discrete Independent Random Variables

Janche Sang
Vernon Rago
John Spragius

Report Number: 92-085
EFFICIENT ALGORITHMS FOR JOINT
DENSITY APPROXIMATIONS OF DISCRETE
INDEPENDENT RANDOM VARIABLES

Janche Sang
Vernon Rego
John Spragins

CSD-TR-92-085
November 1992
Efficient Algorithms for Joint Density Approximations of Discrete Independent Random Variables

Janche Sang  
Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907  

Vernon Rego*  
Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907  

John Spragins  
Dept. of Electrical Engineering  
Clemson University  
Clemson, SC 29634  

Abstract

We present an efficient algorithm to compute the joint density function of a set of discrete independent random variables. The computation is accurate up to the coverage specified. The technique is based on a simple mathematical structure and associated property known as the diamond property. The algorithm generates the product density, or states of the joint random variable, in order of decreasing probability, stopping when a prescribed degree of accuracy has been obtained. Since the algorithm can be seen to exhibit a high degree of inherent parallelism, we propose easily implementable versions of the original algorithm for parallel machines.

*This research was supported in part by NSF award CCR-9102331, NATO award 900108 and the Mathematical Sciences Section of Oak Ridge National Laboratory under contract contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.
1 Introduction

Given a set of \( m \) discrete independent random variables \( X_1, X_2, \ldots, X_m \), where \( X_j \) can take on values from a set of cardinality \( n \), the computation of the density of the joint random variable \( Z = (X_1, X_2, \ldots, X_m) \) requires calculating \( n^m \) probabilities. This computation is prohibitive when either \( m \) or \( n \) is large. Therefore, it is of some interest to develop efficient approximation schemes in situations where numerical values are required. Since the state space of the product density grows exponentially with \( m \), it is not practical to obtain the probability of each state through direct enumeration. Instead, the state probabilities can be generated in an ordered manner, by generating states in order of decreasing contribution to the joint probability mass.

In this paper, we present a novel approach which utilizes a simple mathematical structure and associated property known as the diamond property \([3]\). The algorithm generates state probabilities (and states) in decreasing order, terminating when the probability mass achieves a specified coverage. It is also possible to state generate probabilities in increasing order through a minor modification of the algorithm. In addition, the proposed algorithm has a high degree of inherent parallelism and can be easily parallelized in recursive doubling or in pipelined manner \([2]\) to improve performance.

The approach that we propose for joint probability computation has wide applicability in probabilistic systems, such as in the reliability computation of network systems. In such systems, reliability is computed as a function of a product random variable. For example, in an early study, Chiou and Li \([5]\) developed a method, called the ORDER-M algorithm, to generate the \( k \) most probable states of a network with multimode components. That is, each network component is assumed to operate independently of the other components, with a given probability of being in any one of \( n \) states. One or more of these states is generally associated with operational failure. The authors show that the computational complexity of their algorithm is \( O(m^2 nk + mnk \log k) \). However, the method used by this algorithm to generate new states at each iteration step is not particularly efficient, thus enforcing a total of \( mn \) iterations. In two other studies, Lam and Li \([8]\) and Valvo et al. \([9]\), respectively, present more efficient algorithms by using heap data structures \([1]\). In both cases, the authors restrict
their attention to components that operate in two-modes (i.e., n = 2, or equivalently, binary random variables), thus addressing a particular case of a more general problem. The arguments presented in Valvo et al. [9] are elegant in their use of Hasse diagrams, but are more difficult to appreciate and implement than the arguments based on the elementary diamond property, for such graph based computations.

In this study, we map the domain of the problem to a multigraph, i.e., a graph in which more than one edge may exist between a pair of nodes. We relate the given problem to the problem of finding the k shortest paths between two specific nodes in the multigraph. In another context, the problem of finding the k shortest paths in a simple graph has been studied by Dreyfus [6] and Yen [10]. Because the subject of their studies is a simple graph, their algorithms are not immediately applicable to a multigraph. We use the diamond property to reduce the size of the computation space, thus obtaining an improved time complexity $O(mk \log h)$, where $h$ is the maximum size among all heap sizes occurring during the computation; it is shown that $h$ is always less than $k$, the number of probabilities (or states) required. We point out that the time complexity can be further reduced to $O(mk)$ by using a novel priority queue structure called the calendar queues. In [4], it is shown that insertions and deletions on calendar queues require constant time.

The remainder of the paper is organized as follows. In Section 2 we present the diamond property which underlies our algorithm. We first do this for the case of two random variables (i.e., $m = 2$). In essence, this property allows us to efficiently generate joint probabilities in decreasing order. The general algorithm is presented in Section 3. In Section 4 we show that the algorithm lends itself very well to parallel computation. We proposed two parallel schemes, requiring time complexities of $O(k \log m \log h)$ and $O((m + k) \log h)$, respectively, on a system of $m$ processors. We present a brief conclusion in Section 5.

2 The Diamond Property and Its Application

In this section we introduce the diamond property and apply it to a subproblem, i.e., computing the joint density function of two discrete independent random variables. The restriction is only for ease of exposition, since the general case appears in the next section. The diamond property (see Figure 1(a)) is defined as follows.
Definition 2.1 Let $\rightarrow$ be a binary relation on a set $S$. The relation $\rightarrow$ satisfies the diamond property if for all $L, M, N$ in $S$ such that $L \rightarrow M$ and $L \rightarrow N$, there exists an $O \in S$ such that $M \rightarrow O$ and $N \rightarrow O$.

One example of the utility of the diamond property can be seen in the proof of the Church-Rosser theorem in Lambda Calculus [3]. From the above definition, it should be clear that it can be extended to the transitive closure of a relation. This follows because, at each step, application of the diamond property guarantees that no matter how many transitive steps are taken, distinct paths converge at a common point. Thus, we arrive (see Figure 1(b)) at the following property:

Property 2.1 If a relation $\rightarrow$ satisfies the diamond property, then so does its transitive closure $\rightarrow^*$. 

Now, consider two discrete, independent random variables $X$ and $Y$, with given density functions $P_X()$ and $P_Y()$, respectively. Assume that $X$ and $Y$ are each defined on the state-space \{1, 2, 3, \ldots, n\} and satisfy the conditions:

1. $P_X(X = 1) \geq P_X(X = 2) \geq \cdots \geq P_X(X = n) \geq 0$,
2. $P_Y(Y = 1) \geq P_Y(Y = 2) \geq \cdots \geq P_Y(Y = n) \geq 0$, and
3. $\sum_{x=1}^{n} P_X(X = x) = 1$, $\sum_{y=1}^{n} P_Y(Y = y) = 1$.

The first two conditions given above are easily obtained by renaming states in the $X$ and $Y$ domains, respectively. Let $S$ denote the state-space of the joint random variable $\{X, Y\}$. 

![Figure 1: Diamond property](image)
Observe that each element in $S$ is a joint event of the form \( \{X = x, Y = y\} \), for \( 1 \leq x, y \leq n \). Simply put, each such event may be pictured as one of \( n^2 \) points on the \((x, y)\) coordinate system. The joint probability \( P_{X,Y}(x,y) \) is the value associated with the joint event point \((x, y)\), obtained through independence as a product of marginal probabilities \( P_X(x) \) and \( P_Y(y) \).

For any two elements \( M,N \in S \), we define \( M \geq N \) if \( P_{X,Y}(M) \geq P_{X,Y}(N) \). Thus, it is easy to see that the pair \( S \) and \( \geq \) form a partially ordered set \((S, \geq)\).

As a consequence of ranking state probabilities in the marginal densities of \( X \) and \( Y \), respectively, the state space of the joint density satisfies two useful properties.

**Property 2.2** \( P_{X,Y}(x,y) \geq P_{X,Y}(x,y+1) \) and \( P_{X,Y}(x,y) \geq P_{X,Y}(x+1,y) \), \( 1 \leq x, y \leq n-1 \).

**Proof:** For the first inequality, we use the given condition \( P_X(X = x) \geq P_X(X = x + 1) \), for \( 1 \leq x \leq n - 1 \), to obtain

\[
P_{X,Y}(x,y) = P_X(X = x) \cdot P_Y(Y = y) \geq P_X(X = x + 1) \cdot P_Y(Y = y) = P_{X,Y}(x+1,y)
\]

The second inequality is obtained by symmetry.

**Property 2.3** The relation \( \geq \) satisfies the diamond property on \( S \).

**Proof:** By applying property 2.2 twice, we obtain
Table 1: Density functions for $X$ and $Y$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>0.30</td>
<td>0.25</td>
<td>0.22</td>
<td>0.12</td>
<td>0.11</td>
</tr>
<tr>
<td>$Y$</td>
<td>0.40</td>
<td>0.25</td>
<td>0.15</td>
<td>0.13</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 2: The first 6 largest joint probabilities of $X$ and $Y$

<table>
<thead>
<tr>
<th></th>
<th>(1,1)</th>
<th>(2,1)</th>
<th>(3,1)</th>
<th>(1,2)</th>
<th>(2,2)</th>
<th>(3,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob.</td>
<td>0.1200</td>
<td>0.1000</td>
<td>0.0880</td>
<td>0.0750</td>
<td>0.0625</td>
<td>0.0550</td>
</tr>
</tbody>
</table>

$P_{X,Y}(x, y) \geq P_{X,Y}(x+1, y)$, $P_{X,Y}(x, y+1) \geq P_{X,Y}(x+1, y+1)$.

The utility of the partially ordered set $(S, \succeq)$ becomes clear when it is represented as a directed graph. Each node of the graph represents an element of $S$. In Figure 2 can be seen the diamond property of relation $\succeq$ on $S$. Given two elements $M$ and $N$, an arc exists between them only if the relation $M \succeq N$ holds. In this case, $M$ is said to be the predecessor of $N$, and $N$ is said to be the successor of $M$. For example, element $(2,2)$ is a predecessor of element $(3,2)$ since $(2,2) \succeq (3,2)$. If there is a directed path from element $M$ to element $N$, then the probability of $M$ is greater than or equal to the probability of $N$. This is easy to understand through transitivity, since $M \succeq^* N$. For example, $(1,2) \succeq^* (2,4)$ holds and we have $P_{X,Y}(1,2) \geq P_{X,Y}(2,4)$.

Based on the diamond property, we can efficiently generate the most probable elements in decreasing order. The most probable element is (trivially) obtained as the element whose components have maximum probability in the marginal densities. From this point onwards, more than one candidate will exist for the post of next most probable element. At any stage, an element becomes a candidate for the post of next most probable element if either its only predecessor has, or both its predecessors have, already been selected and inserted into the list of probable elements. Because the number of such candidates may grow, it is necessary to maintain a list of candidates in as efficient a manner as possible, prior to selecting the next
Figure 3: Operation Sequence
most probable element.

The list of most probable elements grows by adding to it, at each step, the next most probable element. At any given step, when the largest element in the current candidate set is selected as the next most probable element, it is deleted from the current candidate set and added to the list of probable elements. At this point, its successors (either one, or both) must be examined for possible candidacy. Each successor that is found to be a candidate must be inserted into the candidate set.

In Figure 3 is shown a descriptive sketch of the procedure. Given the density functions for the two random variables X and Y (see Table 1), the first six elements are generated in the following order: (1,1), (2,1), (3,1), (1,2), (2,2), and (3,2), as shown in Table 2. Starting with element (1,1) which of course corresponds to that with the largest joint probability, the elements (2,1) and (1,2) are inserted into the candidate set. In (b), since $P_{X,Y}(2,1)$ is larger than $P_{X,Y}(1,2)$, element (2,1) is selected as the next most probable element and consequently deleted from the candidate set. Note that only the successor (3,1) of element (2,1) can be inserted to the candidate set. Though element (2,2) is also a successor of (2,1), its violates candidacy because it has another predecessor (1,2) that has not been selected yet. Parts (c) through (f) proceed in a similar manner. Note that in part (f), it is possible that no new candidate can be generated. The above example leads us to conclude that the size of the candidate set is reduced during execution. In particular, the algorithm

- generates elements in order of decreasing joint probability, and
- inserts at most two new candidates into the candidate set, for each new probable element added to the list of probable elements found.

3 Algorithm ENSURE

The approach we take is to map the problem domain to a directed multigraph. The multigraph defined to be is a graph in which more than one edge may exist between two given vertices [7]. Let $G = (V, E)$ be a directed multigraph where $V$ is the set of $m + 1$ vertices, representing the $m$ random variables, $X_1, X_2, \ldots, X_m$, and a source node $0$. The set of edges $E$ is constructed from the state probabilities of each random variable. Here we assume the
probabilities are in decreasing order, i.e., \( P_{X_i}(1) \geq P_{X_i}(2) \geq \cdots \geq P_{X_i}(n) \), for all \( 1 \leq i \leq m \).

Note that a random variable \( X_i \) may have less than \( n \) states. In this situation we may introduce dummy states, with associated probability 0, into the state space of random variable \( X_i \).

In Figure 4 can be seen a transformed graph. Between each node \((i-1)\) and node \(i\), \(1 \leq i \leq m\), there are \( n \) directed edges. Each such edge represents a state probability for the the random variable \( X_i \). Labelling edges from top to bottom, we associate the \( y \)th edge from the top with the probability \( P_{X_i}(y) \). Thus, given a path from node 0 to node \( n \) on the multigraph, the joint probability corresponding to the event represented by that path is simply the product of probabilities along the path. Our task is to design of an efficient algorithm that can be used to generate the paths in order of decreasing probabilities, terminating only when some prescribed coverage is obtained.

The dynamic programming method is used to solve the transformed problem. The principle behind the approach is to first solve the problem for subgraph \( G^i \), involving only nodes 0 through \( i \), and then using the result to solve the problem for subgraph \( G^{i+1} \). The exposition in Section 2 should help in demonstrating the solution scheme for subgraph \( G^2 \).

Let

\[
D_i^k := \text{the product of the probabilities along the } k\text{th largest path from node 0 to node } i,
\]

and

\[
d_i[z,y] = \text{the product of the probabilities along a path which is composed of the } z\text{th largest subpath from node 0 to node } (i-1) \text{ and the } y\text{th largest edge between node } (i-1) \text{ and node } i.
\]
That is,
\[ d_i[x, y] = D_{i-1} \ast P_{X_i}(y) \]  
\[ (1) \]

**Property 3.1** \( d_i[x, y] \geq d_i[x, y + 1] \) and \( d_i[x, y] \geq d_i[x + 1, y] \), for \( 1 \leq i \leq m \).

**Proof:** This follows from Equation 1 and the facts that \( D_i^x \geq D_i^{x+1} \) and \( P_{X_i}(y) \geq P_{X_i}(y + 1) \).

It is helpful to observe that Property 3.1 is similar to Property 2.2. Note that \( D_i \) is essentially a random variable with density function, say \( \{D^k_i : 1 \leq k \leq n_i\} \), identical to the joint density of \( X_1, X_2, \ldots, X_i \). Therefore, by applying the diamond property discussed in Section 2, we may generate \( D_{i+1} \) from \( X_{i+1} \) and the previously computed \( D_i \). Note that it is *not necessary* to generate all the probabilities of \( D_i \) before computing \( D_{i+1} \). Instead, for some \( k \geq 1 \), once \( D_i^k \) is generated, the computation of \( D_{i+1}^k \) can proceed.

In Figure 5 is given an algorithm which ensures (hence the name) that, upon termination, the computed quantities \( D_j^i \), \( 1 \leq j \leq k \), are the \( k \) largest joint probabilities whose summation is guaranteed to achieve the desired coverage.

**Implementation issues:**

For computing \( D_i \), \( 2 \leq i \leq m \), it is necessary to maintain \( m - 1 \) candidate sets \( \mathcal{N}_i \) in our algorithm. The two operations, \( \text{EXTRACTMAX} \) and \( \text{INSERT} \), are required for operating on these sets (step 5 and 7). \( D_1 \) is initialized to be \( X_1 \), except for those \( D_1^k, k \geq n \) that are set zero. To print out the state corresponding to a joint probability, an array \( B[i], 1 \leq i \leq m \) is used as a buffer to record the state (i.e. \( y \)) of each variable (see step 6 and 9). To check whether an element is a potential candidate (step 7), we may use \( m - 1 \) boolean matrices \( f_i \) (not shown in the algorithm). Once a candidate \( d_i[x, y] \) is selected in step 5, \( f_i[x, y] \) is set to be \( \text{TRUE} \). So an element can become a candidate only if its predecessor(s) is (are) marked \( \text{TRUE} \).
Algorithm ENSURE:

Input: Number of random variables $m$, number of values $n$ each variable can have, the probability matrix $(m \times n)$ associated with those random variables, and coverage $\alpha$.

Output: The joint probabilities in decreasing order until the coverage is obtained.

1. (* Initialization *)
   - for $k \leftarrow 1$ to $m$ do
     - $D^k_1 = P_{X_1}(k)$;
     - for $i \leftarrow 2$ to $m$ do
       - begin (* initialize heaps and generate the largest joint probability *)
         - $D^i_1 \leftarrow D^i_{i-1} \times P_{X_i}(1)$;
         - $H_i \leftarrow \{d_i[1,1]\}$;
       - end; (* for *)
     - OUTPUT $D^1_1$;
   - $k \leftarrow 2$; $sum \leftarrow 0.0$;

2. while ($sum < \alpha$) do
   - begin
     - for $i \leftarrow 2$ to $m$ do
       - begin
         - $D^i_k \leftarrow EXTRACT\, MAX(H_i)$; (* let the element currently selected be $d_{i,x,y}$ *)
         - $B[i] \leftarrow y$; (* record $y$ for the $i$th variable *)
         - generate two new candidates if they are qualified:
           - $d_{i,x+1,y} \leftarrow D^i_{i-1} \times P_{X_i}(y)$;
           - $d_{i,x,y+1} \leftarrow D^i_{i-1} \times P_{X_i}(y+1)$;
         - $H_i \leftarrow H_i \cup \{d_{i,x+1,y},d_{i,x,y+1}\}$;
       - end; (* for *)
     - (* output the $k$th joint probability *)
     - for $i \leftarrow 2$ to $m$ do
       - OUTPUT $B[i]$;
     - OUTPUT "k=",$k$, "Joint Prob.=",$D^k_m$
     - $k = k + 1$; $sum = sum + D^k_m$
   - end. (* while *)

Figure 5: Algorithm ENSURE
Analysis of Algorithm ENSURE:

For efficiency, the set \( H_i \) should be implemented as a heap. The operations \( \text{EXTRACT MAX} \) (\( \text{EXTRACT MIN} \)) and \( \text{INSERT} \) have a time complexity of \( \log(n) \), for a heap of size \( n \).

Let \( h_i \) denote the maximum size that heap \( H_i \) grows to during execution, and recall that \( k \) is the number of probable states to be generated. Defining \( h = \max(h_i) \) for \( 2 \leq i \leq m \), we have the property that:

**Property 3.2** \( h \leq k \).

**Proof:** In the inner loop of the algorithm (step 4 to step 8), each time an element is extracted from the heap, at most two new elements can be inserted. Therefore, the size of each heap increases by at most one element at each iteration step. Since there are \( k \) iterations in all, \( h_i \leq k \) for every \( i \). \( \square \)

**Theorem 3.1** Algorithm ENSURE ensures that the desired coverage is obtained in time \( O(mk \log h) \).

**Proof:** The check of candidate qualification in step 7 can be done in constant time. In addition, from the previous property, we know that the inner loop costs \( O(m \log h) \) in the worst case. Thus, the time complexity of the algorithm is \( O(mk \log h) \) since we need to iterate \( k \) times to ensure desired coverage. \( \square \)

A more efficient data structure than the heap, called the calendar queue, has recently been proposed. This structure allows the operations of \( \text{EXTRACT MIN} \) and \( \text{INSERT} \) to be done in constant time. It is a very simple data structure which is similar to a multiple list but does not require an overflow list. Using such calendar queues instead of heaps, the time complexity of our algorithm may be improved to \( O(mk) \).

An alternative approach is to use a divide-and-conquer method instead of dynamic programming. First, we define

\[ D_{i,j}^k = \text{the product of the probabilities along the } k\text{th largest path from node } i \text{ to node } j. \]

We proceed by cutting the multigraph into two halves, and calculating the \( k \) largest joint probabilities of each, i.e. \( D_{0/2/m}^k, D_{2/m+1}^k \). We next combine these values, using the diamond
property. Each half of the multigraph can be cut again recursively. Let \( T(m) \) be the time required to solve the problem. We have the following recurrence relation:

\[
T(m) = \begin{cases} 
   c & m = 1, \\
   2T(m/2) + k \log h & m > 1.
\end{cases}
\]

Despite the simple description, this algorithm still requires the \( O(mk \log h) \) time of the previous algorithm, since the combination part needs \( O(k \log h) \) time, and there are \( m - 1 \) combination operations.

Though the dynamic programming approach and the divide-and-conquer method make no difference in terms of complexity, the complexity of their parallel versions are greatly affected. In the next section, we propose two parallel algorithms based on these methods.

4 Parallel Algorithms

There are two different approaches to parallelizing the enumeration of the requisite set of joint probabilities. One approach is to use a pipelined version of the sequential dynamic programming scheme. The other approach is to adopt a recursive doubling scheme for the sequential divide-and-conquer version. Both approaches have been applied in parallelizing various sequential algorithms[2]. We now examine how these methods may be used to obtain parallel versions of algorithm ENSURE. We assume that the number of available processors on the underlying parallel architecture is \( m \).

**Pipelining:**

The pipelining strategy exploits temporal parallelism. The computation is divided into a number of steps called stages. Each processor processes a single stage at a time, essentially filtering input data from a single predecessor, and sending the filtered output data to a single successor. The algorithm ENSURE can be seen to possess a high degree of inherent parallelism in steps 3 and 4. During the \( k \)th iteration, each processor \( i \) computes the joint probability \( D_i^k \) and sends this quantity to its successor, processor \( i + 1 \). Processor \( i \) then waits to receive quantity \( D_{i-1}^k \) from processor \( i - 1 \) so that it may generate new candidates. Distinct processor computations can be overlapped through pipelining.
Algorithm Parallel ENSURE:

1. (* get processor id *)
   \[ i \leftarrow \text{getprocessorid}(); \]
2. if \((i == 1)\) then
   \[
   \text{begin}
   \text{for } k \leftarrow 1 \text{ to } n \text{ do}
   \text{begin}
   D_i^k \leftarrow P_{X_i}(k);
   \text{send}(2, D_i^1);
   \text{end;}
   \text{end; (* if *)}
   \]
3. \( k \leftarrow 2; \) \( \text{sum} \leftarrow 0.0; \)
4. while \((\text{TRUE})\) do
   \[
   \text{begin}
   \text{if } (i == 1) \text{ then do}
   \text{send}(2, D_i^1);
   \text{else begin}
   \text{begin}
   D_i^k \leftarrow \text{EXTRACTMAX}(H_i); (* \text{let the element currently selected be } d_i[x, y] *)
   B[i] \leftarrow y;
   \text{send}(i + 1, D_i^k);
   \text{receive}(i - 1, D_{i-1}^k);
   \text{generate two new candidates if they are qualified:}
   d_i[x + 1, y] \leftarrow D_{i-1}^{k+1} \times P_{X_i}(y);
   d_i[x, y + 1] \leftarrow D_{i-1}^{k+1} \times P_{X_i}(y + 1);
   H_i \leftarrow H_i \cup \{d_i[x + 1, y], d_i[x, y + 1]\};
   \text{end; (* if *)}
   \text{end; (* if *)}
   \text{end; (* while *)}

\text{Figure 6: Algorithm Parallel ENSURE using pipelining}
The generation of the largest joint probability is done in $m$ distinct stages, where each stage requires a time complexity of $O(\log n)$. Following the first set of $m$ stages at the end of which the largest joint probability is available, a single extra stage is sufficient for the next largest joint probability. In this manner, at the end of each stage after the first $m$, a new joint probability is obtained, in order of decreasing value. Thus, generation of the $k$ largest probabilities can be obtained in a total of $m$ stages (for the largest probability) plus $k$ additional stages (for the remaining probabilities). Therefore, the time complexity of the pipelined version, using $m$ processors, reduces to $O((m + k)\log n)$ (or an even better $O(m + k)$ if calendar queues are used instead of heaps).

The code for the parallel version of algorithm ENSURE can be seen in Figure 6. The function $send(i, data)$ is used by the processor executing the send function to send data to some processor $i$. Similarly, the function $receive(i, data)$ is used by the processor executing the receive function to receive data from some processor $i$. The algorithm is terminated by the last processor, i.e., processor $m$, in the pipeline, when it determines that a prescribed coverage has been achieved. A description of the pattern of communication between processors in the pipelined scheme is shown in Figure 7.

**Recursive doubling:**

In the previous section it was shown that the generation of a joint probability is efficiently done through a divide-and-conquer approach. The technique of recursive doubling exploits parallelism within the divide-and-conquer strategy. For example, in the first stage, the distinct quantities $D_{1,2}^k, D_{3,4}^k, \ldots, D_{m-1,m}^k$ can be computed simultaneously at the distinct processors 1, 3, $\ldots$, $m - 1$ respectively. In the second stage, these processors can generate the quantities $D_{1,4}^k, D_{5,8}^k$, etc., so that the total number of stages required for the computation of the single
The joint probability $D_{i,m}^k$ is $O(\log m)$. Since a total of $k$ joint probabilities are required, the time complexity of the parallel algorithm becomes $O(k \log m \log h)$. A graphical description of the recursive doubling scheme as applied to our problem can be seen in Figure 8.

5 Conclusion

In contrast to previous studies which address particular versions of the general problem, use of the diamond property led easily to heap based data structures for the computation of product densities. In particular, it seemed to us to be a lot simpler to use than Hasse diagrams, in terms of the number of nodes and arcs on the underlying graphs. Keeping the size of the new candidate set to be added to the existing set to at most two appears to be one advantage of using the diamond property. The other advantage is the natural generation of ordered joint probabilities.

Given the inherent parallelism of the original algorithm, the pipelined and recursive doubling schemes follow quite naturally. Thus, an already efficient sequential computation of time complexity $O(mk \log h)$, based on heaps, can be reduced to $O(mk)$ by using calendar queues. Using a multiprocessor with $m$ processors allows a further reduction in complexity to $O((m + k) \log h)$ (with pipelining) and $O(k \log m \log h)$ (with recursive doubling).

It would be of some interest to consider equivalent computations in the case where the underlying random variables are dependent. Clearly, the structure of the dependence will affect the computation to no small extent. Another possible generalization is the extension of the two-dimensional diamond property to a multi-dimensional diamond property, and its
application to multi-dimensional generalizations of the original problem.

References


