An Efficient Symbolic Out-of-Core Solution Method for Markov Models[*]

Rashid Mehmood, David Parker, and Marta Kwiatkowska

School of Computer Science, University of Birmingham,
Birmingham B15 2TT, United Kingdom
{rxm,dxp,mzk}@cs.bham.ac.uk

Abstract. In recent years, disk-based approaches to the analysis of Markov models have proved to be an effective method of combating the state space explosion problem. Coupled with parallel and symbolic techniques, disk-based methods have demonstrated impressive performance for numerical solution. In an earlier paper, we presented a novel, symbolic out-of-core algorithm which used MTBDD-based data structures for matrix storage in RAM and disk-based storage for solution vectors. This extended the size of models which could be solved on a standard workstation. This paper reports on a significant improvement to our earlier work obtained by using an alternative scheme for decomposing the matrix into blocks. We present experimental results for three benchmark models, a Kanban manufacturing system, a cyclic server polling system and a flexible manufacturing system, and analyse the performance of our implementation. In general, we double the speed of numerical solution. We report results for models as large as 216 million states and 2.1 billion transitions on a workstation with 512MB RAM.

1 Introduction

Continuous-time Markov chains (CTMCs) are a widely used model for the performance evaluation of communication networks and computer systems. A typical analysis of a CTMC requires computation of its *steady-state probabilities*, a problem which reduces to solving a linear equation system of size equal to the CTMC. Unfortunately, the size of these models usually grows exponentially with the number of parallel components they contain, a phenomenon called the *state space explosion problem*. A range of techniques exist to combat this problem. They can be broadly classified into *explicit* approaches, which use a data structure of size proportional to the number of states and transitions in the CTMC, and *implicit* approaches, where this explicit storage is avoided. The latter includes *symbolic* (BDD-based) methods [16, 21, 6, 10, 22, 12, 29], *on-the-fly* methods [19] and *Kronecker* methods [31, 13, 8, 7]. Another classification is between *in-core* approaches, where data is stored in the main memory of a computer, and *out-of-core* approaches, where it is stored on disk. Both serial [17, 18] and

[*] Supported in part by EPSRC grants GR/S11107 and GR/N22960

parallel [24, 5] implementations of out-of-core techniques have been presented. Many standard parallel approaches also exist [11, 1, 9, 15].

The above methods provide a range of cures for the matrix storage problem. However, all approaches are hindered by storage of the probability vector(s) needed during the numerical solution phase, either because symbolic (BDD-based) representations of these vectors were inefficient or because explicit representations were used. In [26], an explicit out-of-core method was presented which used disk-based, explicit storage of both the probability vector and the CTMC matrix, thus relaxing these memory limitations and increasing the size of solvable model.

In [27], this work was adapted to form a *symbolic out-of-core algorithm*, the idea being to store the CTMC matrix symbolically, in-core, and to store the probability vector explicitly, on disk. For this purpose, the probability vector is divided into a number of blocks. The blocks are read from disk as required, updated, and then written back to the disk. Generally, by increasing the number of blocks, the total memory required can be reduced, albeit at a cost of lengthened run times. This increases the size of model which can be solved on a single workstation and is also amenable to parallelisation. In [27], the symbolic matrix storage scheme used was *offset-labelled MTBDDs* [29, 30]. In fact, this technique should be equally applicable to other implicit storage solutions, such as matrix diagrams [12] or Kronecker representations.

This paper presents an improvement to our symbolic out-of-core approach, obtained by using an alternative block decomposition of the matrix which is better suited to the symbolic matrix storage scheme used. Consequently, it results in a significant speedup for the matrix-vector product (MVP) operation, which is the core component of the numerical solution phase. We give experimental results for our implementation, as applied to a Kanban manufacturing system [13], a flexible manufacturing system (FMS) [14] and a cyclic server polling system [23]. We typically double the speed of numerical solution.

The paper is organised as follows. In the rest of this section, we give a short summary of the use of out-of-core methods in stochastic modelling. Section 2 deals with the iterative solution techniques which we use in this paper. Section 3 describes the MTBDD data structure, which we use for representing CTMCs. Section 4 gives a detailed description of our symbolic out-of-core algorithm. In Section 5, we present and analyse experimental results from our implementation. Finally, Section 6 concludes the paper.

A History of Out-of-Core Solutions in Stochastic Modelling.
Out-of-core algorithms have been in use for a long time. By *out-of-core algorithms*, we mean those which are designed to achieve high performance when their data structures are stored on disk. A survey of such techniques for numerical linear algebra can be found in, for example, [36].

In performance evaluation, Deavours and Sanders [17, 18] were the first to use an out-of-core technique for the steady-state solution of Markov models. They solved a 15 million state system on a workstation with 128MB RAM.

Since then, some excellent developments have been seen. The approach of [17] was parallelised in 1999 [24], and solutions of up to 50 million state systems on a 16-node Fujitsu parallel computer were reported. Bell and Haverkort [5] extended the size of solvable models further in 2001 and reported the parallel out-of-core solution of a 724 million state system on a 26-node dual-processor cluster. Subsequently, [26,27] focused on the out-of-core storage of the vector, and extended these limits even further. The solution of a 41 million state system on a 128MB RAM machine was demonstrated in [26]. In [27], by combining symbolic and out-of-core approaches, solution of a 133 million state system on a single workstation was reported.

2 Iterative Methods for CTMC Solution

Performance measures of interest for stochastic models are traditionally derived by generating and solving a Markov chain obtained from some high-level formalism. We focus in this paper on the computation of steady-state probabilities for a CTMC. If $Q \in \mathbb{R}^{n \times n}$ is the CTMC's infinitesimal generator matrix, and $\pi(t) = [\pi_1(t), \pi_2(t), \ldots, \pi_n(t)]$ is the transient state probability row vector, where $\pi_i(t)$ denotes the probability of the CTMC being in state i at time t, then the steady-state probability vector is defined as $\pi = \lim_{t \to \infty} \pi(t)$. It can be computed by solving the following linear equation system:

$$\pi Q = 0, \ \sum_{i=0}^{n-1} \pi_i = 1. \tag{1}$$

The order of the infinitesimal generator matrix Q equals the number of states, n, in the CTMC. The off-diagonal elements of Q satisfy $q_{ij} \in \mathbb{R}_{\geq 0}$, and the diagonal elements are given by $q_{ii} = -\sum_{j \neq i} q_{ij}$. The matrix Q is usually very sparse; further details about the properties of these matrices can be found in [35]. Equation (1) can be reformulated as $Q^T \pi^T = 0$, and well-known methods for the solution of systems of linear equations of the form $Ax = b$ can be used.

Numerical solution methods for linear systems of the form $Ax = b$ are broadly classified into *direct* methods and *iterative* methods. Direct methods attempt to compute the exact solution to a linear system in a finite number of steps. Iterative methods begin with an initial approximate solution and then use successive approximations to obtain increasingly more accurate solutions at each step until a required accuracy is achieved [4]. For large systems, direct methods become impractical due to the phenomenon of *fill-in*, caused by the generation of new matrix entries during the factorisation phase. Iterative methods, however, do not modify the matrix A; rather, they involve the matrix only in the context of matrix-vector product (MVP) operations. See [20] for further information on direct methods and [4,34,35] for iterative methods.

In this paper, we consider *stationary* iterative methods, those which can be expressed in the simple form $x^{(k)} = Fx^{(k-1)} + c$, where $x^{(k)}$ is the approximation to the solution vector at the k-th iteration and neither F nor c depend on k [4].

In the k-th iteration of the Jacobi method, for example, we calculate:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i \; - \; \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right), \tag{2}$$

for $0 \leq i < n$, where a_{ij} denotes the element in row i and column j of matrix A and the term $x_i^{(k)}$ indicates the i-th element of the k-th iteration vector. The above equation can also be written in matrix notation as:

$$x^{(k)} = D^{-1}(L+U)\, x^{(k-1)} \; + \; D^{-1}b, \tag{3}$$

where $A = D - (L+U)$ is a partitioning of A into its diagonal, lower-triangular and upper-triangular parts, respectively. Note the similarities between $x^{(k)} = Fx^{(k-1)} + c$ and (3) above. The Jacobi method can be formulated into an MVP (*matrix-vector product*) based algorithm, a typical iteration of which can be implemented as follows:

```
1. $\tilde{x} \leftarrow b$
2. $\tilde{x} \leftarrow \tilde{x} - \breve{A}x$
3. $\tilde{x} \leftarrow D^{-1}\tilde{x}$
4. Test for convergence, stop if converged
5. $x \leftarrow \tilde{x}$
```

where $\breve{A}$ contains the off-diagonal elements of matrix A, i.e. $\breve{A} = -(L+U)$. Observe the similarity between this algorithm and (3) above. Line 2 of the algorithm performs the MVP operation. The algorithm requires storage for two iteration vectors (the previous iterate x and the new iterate $\tilde{x}$), for the matrix $\breve{A}$ and for the diagonal entries in D. Note that the new approximation of the solution vector is calculated using only the old approximation of the solution. This makes the Jacobi method well suited for parallelisation, but means it tends to exhibit slow convergence.

The Gauss-Seidel method, which in practice converges faster than the Jacobi method, uses the most recently available approximation of the solution:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i \; - \; \sum_{j<i} a_{ij} x_j^{(k)} \; - \; \sum_{j>i} a_{ij} x_j^{(k-1)} \right) \tag{4}$$

for $0 \leq i < n$. The other advantage of the Gauss-Seidel algorithm is that it can be implemented using only one iteration vector. The Gauss-Seidel method can also be expressed in matrix notation:

$$x^{(k)} = (D-L)^{-1}\, U\, x^{(k-1)} + (D-L)^{-1}\, b \tag{5}$$

where D, L and U are as described for the Jacobi method above. In practice, it would be inefficient to perform Gauss-Seidel in this way due to the computation required for matrix inverses.

```
1.  for $p = 0$ to $P - 1$
2.      $\tilde{X}_p \leftarrow B_p$
3.      for $q = 0$ to $P - 1$
4.          $\tilde{X}_p \leftarrow \tilde{X}_p - \breve{A}_{pq} X_q$
5.      end for
6.      $\tilde{X}_p \leftarrow D_{pp}^{-1} \tilde{X}_p$
7.      Test for convergence
8.      $X_p \leftarrow \tilde{X}_p$
9.  end for
10. Stop if converged
```

Fig. 1. An iteration of the pseudo Gauss-Seidel algorithm

2.1 The Pseudo Gauss-Seidel Method

In the Jacobi method, the order in which matrix entries are accessed within a single iteration is unimportant. For Gauss-Seidel, access to individual rows is required. For these reasons, symbolic implementations of iterative methods based on MTBDDs are better suited to Jacobi than Gauss-Seidel. Parker [30] resolves this problem by introducing the *pseudo Gauss-Seidel* method, a compromise between Jacobi and Gauss-Seidel, which can be summarised as follows.

Let the state space S of the CTMC be divided into P contiguous partitions $S_0, \ldots, S_{P-1}$ of sizes $n_0, \ldots, n_{P-1}$, such that $n = \sum_{i=0}^{P-1} n_i$. We make no assumptions about the relative sizes of these partitions. Using this, the matrix A can be divided into P^2 blocks, $\{A_{pq} \mid 0 \leq p, q < P\}$, where the rows and columns of block A_{pq} correspond to the states in S_p and S_q, respectively, i.e. block A_{pq} is of size $n_p \times n_q$. We introduce the additional notation $N_p = \sum_{i=0}^{p-1} n_i$, for $0 \leq p \leq P$. A partition S_p includes states with indices N_p up to $N_{p+1} - 1$. We also define $n_{\max} = \max\{n_p \mid 0 \leq p < P\}$. Finally, we denote by $block(i)$ the index of the block containing state i, i.e. the unique $0 \leq p < P$ such that $N_p \leq i < N_{p+1}$. The k-th iteration of the pseudo Gauss-Seidel method comprises the computation:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < N_{block(i)}} a_{ij}\, x_j^{(k)} - \sum_{j \geq N_{block(i)} \,\wedge\, j \neq i} a_{ij}\, x_j^{(k-1)} \right) \tag{6}$$

for $0 \leq i < n$. This can be written in block notation form as:

$$X_p^{(k)} = D_{pp}^{-1} \left(B_p - \sum_{q<p} \breve{A}_{pq}\, X_q^{(k)} - \sum_{q \geq p} \breve{A}_{pq}\, X_q^{(k-1)} \right) \tag{7}$$

for $0 \leq p < P$, where $X_p^{(k)}$, $X_p^{(k-1)}$ and B_p are the p-th blocks of vectors $x^{(k)}$, $x^{(k-1)}$ and b respectively, D and $\breve{A}$ contain the diagonal and off-diagonal elements of A respectively, and as above, F_{pq} denotes the (p, q)-th block of a matrix F.

The pseudo Gauss-Seidel method can be formulated into an MVP-based algorithm, a typical iteration of which is shown in Figure 1. The algorithm works as follows. Each iteration is divided into P phases. In the p-th phase, the method updates elements in the p-th block of the solution vector. It does this using the most recent approximation for each element of the solution vector available, i.e. it uses values from the previous iteration for entries corresponding to vector blocks $p, \ldots, P-1$ and values from earlier phases of the current iteration for entries corresponding to blocks $0, \ldots, p-1$. The pseudo Gauss-Seidel method can be related to the Jacobi and Gauss-Seidel methods by considering Jacobi to be the case where $P = 1$ and Gauss-Seidel to be the case where $P = n$.

Note that the p-th phase of an iteration, which computes the p-th block of the solution vector, only requires access to entries from the p-th row of blocks in $\check{A}$, i.e. $\check{A}_{pq}$ for $0 \leq q < P$. We illustrate this in Figure 2 for $P = 4$ and $p = 1$. Here, all blocks are of equal size but this is generally not the case. The matrix and vector blocks used are shaded grey. A *unit of computation* comprises the multiplication of a single matrix block by a single vector block (a *sub-MVP*). This corresponds to line 4 of the algorithm in Figure 1. As we will see later, sub-blocks of matrices represented as MTBDDs are particularly easy to access. This is one of the main motivations for introducing the pseudo Gauss-Seidel method.

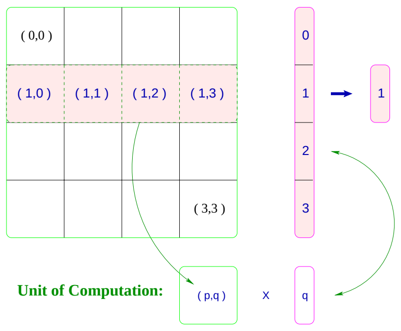


Fig. 2. Matrix vector multiplication at block level

The algorithm given in Figure 1 requires, in addition to the matrix storage, one iteration vector x of size n to store the solution vector, the p-th block of which is denoted X_p, and another vector $\tilde{X}_p$ of size $n_{\max}$ to accumulate the sub-MVPs. The subscript p of $\tilde{X}_p$ in the algorithm is used to make the description intuitive and to keep the vector block notation consistent; it does not imply that we have used P such arrays.

Since pseudo Gauss-Seidel uses some elements of the most recent approximation in each iteration, it generally converges faster than the Jacobi method. Factors which will affect the speed of its convergence include the number of partitions, P, and the sizes of these partitions. We can show, though, that the convergence characteristics of pseudo Gauss-Seidel are similar to those of Jacobi and Gauss-Seidel, as presented for example in [35].

All three iterative methods can be associated with a *splitting* of the matrix $A = M - N$. An iteration of each method can then be written as $x^{(k)} = M^{-1}Nx^{(k-1)} + M^{-1}b$. For Jacobi, M contains the diagonal entries of A; for Gauss-Seidel, M contains the diagonal and lower-triangular entries of A; and for pseudo Gauss-Seidel, we define M as:

$$M_{ij} = \begin{cases} A_{ij} & \text{if } j < N_{block(i)} \text{ or } j = i \\ 0 & \text{otherwise.} \end{cases}$$

In all three cases, N is equal to $M - A$. In [35], the convergence properties of both Jacobi and Gauss-Seidel are analysed for the specific case of computing steady-state probabilities for a CTMC. This analysis uses the fact that both M^{-1} and N are non-negative, which is shown to be true if M can be obtained from A only by setting off-diagonal elements to zero. As can be seen above, this is also true for pseudo Gauss-Seidel.

In all the experiments presented in this paper, we have used the following relative error criterion:

$$\max_i \left(\frac{\mid x_i^{(k)} - x_i^{(k-1)} \mid}{\mid x_i^{(k)} \mid} \right) \ < \ 10^{-6}. \tag{8}$$

3 MTBDD-Based CTMC Storage

MTBDDs (multi-terminal binary decision diagrams) [16, 2] are an extension of BDDs (binary decision diagrams). An MTBDD is a rooted, directed acyclic graph which represents a function mapping Boolean variables to real numbers. MTBDDs can be used to represent real-valued vectors and matrices by encoding their indices as Boolean variables. Standard operations such as matrix-vector multiplication can then be implemented efficiently on the data structure. It has been shown in [21, 3, 22] how such operations can then be used to implement iterative numerical solution techniques including the Power and Jacobi methods.

The principle reason for using an MTBDD representation is that it can provide extremely compact storage of large matrices, provided that structure and regularity derived from their high-level description can be exploited. However, the performance of early MTBDD-based implementations of numerical computation was typically found to be poor, especially in comparison to traditional, explicit implementations based on sparse matrices and arrays. The drawback of the latter, though, is that they can be expensive in terms of memory.

In [29, 30], a hybrid approach was presented, representing the matrix as an MTBDD and the solution vector as an array. This was achieved by making modifications to the MTBDD, labelling nodes with integer offsets. The entries of the matrix can then be extracted by traversing the nodes and edges of the graph and using the offsets to calculate their row and column indices. The modified data structure is called an *offset-labelled MTBDD*. It was found that this approach retained the compact storage advantages of MTBDDs and, for typical examples, could almost match the numerical solution speed of sparse matrices.

Figure 3 shows a small example of a matrix $\breve{A}$, as might be used in the iterative numerical solution techniques described in the previous section, and its representation as an offset-labelled MTBDD. Figure 4 gives a table explaining how the information is encoded. Row and column indices of the matrix are encoded using Boolean variables (x_1, x_2) and (y_1, y_2), respectively. This is demonstrated in the first three columns of the table. An entry of the matrix can be read from the MTBDD by tracing a path from top to bottom, at each node taking an *else* edge (dashed line) or *then* edge (solid line) if the variable labelling the node is 0 or 1 respectively. The value read off at the bottom of the path is equal to the entry of the matrix, as illustrated by the fourth column of the table. The fact that the variables for rows and column indices are interleaved is a common variable ordering heuristic in BDD-based representations to reduce graph size. The integer values on the nodes are used to compute the row and column indices of matrix entries in terms of reachable states only. This is typically essential since the *potential* state space can be much larger than the *actual* state space. See [29, 30] for more information. All entries of the matrix can be extracted in a single recursive traversal of this data structure.

$$\breve{A} = \begin{pmatrix} 0 & 0.9 & 0 & 1.2 \\ 1.2 & 0 & 0 & 1.2 \\ 0 & 0 & 0 & 0 \\ 3.7 & 0.9 & 0 & 0 \end{pmatrix}$$

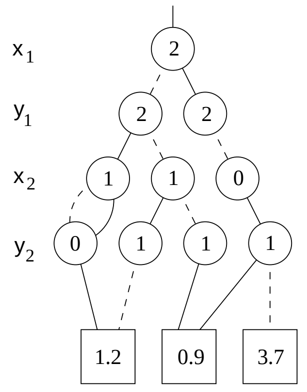


Fig. 3. A matrix $\breve{A}$ and an offset-labelled MTBDD representing it

Entry of $\breve{A}$	Encoding				MTBDD Path				
	x_1	x_2	y_1	y_2	x_1	y_1	x_2	y_2	
$(0,1) = 0.9$	0	0	0	1	0	0	0	1	$\rightarrow 0.9$
$(0,3) = 1.2$	0	0	1	1	0	1	0	1	$\rightarrow 1.2$
$(1,0) = 1.2$	0	1	0	0	0	0	1	0	$\rightarrow 1.2$
$(1,3) = 1.2$	0	1	1	1	0	1	1	1	$\rightarrow 1.2$
$(3,0) = 3.7$	1	1	0	0	1	0	1	0	$\rightarrow 3.7$
$(3,1) = 0.9$	1	1	0	1	1	0	1	1	$\rightarrow 0.9$

Fig. 4. Table explaining the representation of matrix $\breve{A}$ in Figure 3

Note that MTBDDs are an inherently recursive data structure: each node is itself an MTBDD. Furthermore, each node can be seen to represent a submatrix of the matrix represented by the whole MTBDD. For example, descending one level of the MTBDD in Figure 3 (i.e. a pair of $\mathsf{x_i}$ and $\mathsf{y_i}$ variables), we see that each $\mathsf{x_2}$ node represents a 2×2 submatrix of the 4×4 matrix $\breve{A}$. This allows convenient and fast access to individual submatrices.

4 A Symbolic Out-of-Core Solution with MTBDDs

In an implementation of the iterative algorithms mentioned in Section 2, the matrix can be stored using a sparse storage scheme (see for example [4, 35, 34]) and the vector(s) can be stored as an array of doubles. However, this makes the total memory requirements for large CTMCs well above the size of the RAM available in standard workstations. One possible solution [18] is to store the matrix on disk and read blocks of matrix into RAM when required. The in-core storage of iteration vector(s), however, can still be prohibitive. This problem can be solved by either keeping the vector on disk [26] or by distributing the vector among multiple processors [24, 5].

The out-of-core scheduling of both the matrix and the iteration vector incurs a huge penalty in terms of disk I/O. Keeping the matrix in-core, in a compact representation can significantly reduce this penalty. This motivates our symbolic out-of-core solution. The idea is to keep the matrix in-core, in an appropriate symbolic data structure, and to store the probability vector on disk. The iteration vector is divided into a number of blocks. During the iterative computation phase, these blocks can be fetched from disk, one after another, into main memory to perform the numerical computation. We have used offset-labelled MTBDDs [29, 30] for CTMC storage, while the iteration vector for numerical computation is kept on disk as an array.

In Section 4.1, we present our symbolic out-of-core algorithm, focusing on the high-level issues related to the out-of-core implementation of the iterative algorithm. In Section 4.2, we explain the implementation of the sub-MVP operation.

4.1 The Out-of-Core Algorithm

As described in Section 2, we reformulate the system $\pi Q = 0$ to $Q^T \pi^T = 0$ and solve $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. Furthermore, we define $\breve{A}$ to be the matrix containing the off-diagonal entries of the matrix A, and d to be a vector containing the diagonal entries of A. The iterative method we use to solve the system $Ax = 0$ is the pseudo Gauss-Seidel method, which has been explained in Section 2.1. An MVP-based algorithm for a typical iteration was given in Figure 1.

A high-level description of the symbolic out-of-core algorithm for the numerical solution of the linear system $Ax = 0$ is shown in Figure 5. The algorithm is based on that of our earlier approaches [26, 27]. We include the new version in

Integer constant: P (*number of blocks*)
Semaphores: S_1, S_2: occupied
Shared variable: *Dbox (to read diagonal blocks into RAM)*
Shared variables: $Xbox_0$, $Xbox_1$ *(to read solution vector x blocks into RAM)*

Disk-IO Process

```
 1. Local variable: $p$, $q$, $k$, $m$
 2. $m \leftarrow P - 1$
 3. while not converged
 4.   for $p = 0$ to $P - 1$
 5.     $k \leftarrow 1$
 6.     for all non-zero blocks $\breve{A}_{pq}$
 7.       if $k = 0$
 8.         read $X_q$ from disk
 9.       end if
10.       Signal($S_1$)
11.       Wait($S_2$)
12.       if $k \neq 0$
13.         read $D_p$ into Dbox
14.         write $X_m$ to disk
15.         $k \leftarrow 0$
16.       end if
17.     end for
18.     $m \leftarrow p$
19.   end for
20. end while
```

Compute Process

```
 1. Local variable: $p$, $q$, $\tilde{X}_p[\,]$
 2. while not converged
 3.   for $p = 0$ to $P - 1$
 4.     $\tilde{X}_p \leftarrow 0$
 5.     for all non-zero blocks $\breve{A}_{pq}$
 6.       Wait($S_1$)
 7.       Signal($S_2$)
 8.       $\tilde{X}_p = \tilde{X}_p - \breve{A}_{pq} X_q$
 9.     end for
10.     $\tilde{X}_p \leftarrow D_p^{-1}\,\tilde{X}_p$
11.     Test for convergence
12.   end for
13. end while
```

Fig. 5. The symbolic out-of-core pseudo Gauss-Seidel iterative algorithm

full for completeness. The algorithm is implemented using two separate concurrent processes: the *Disk-IO Process* and the *Compute Process*. The two processes communicate via shared memory and synchronise with semaphores.

The algorithm of Figure 5 assumes that the CTMC matrix to be solved is stored in-core, using the offset-labelled MTBDD data structure. The matrix is divided into P^2 blocks, where blocks can be unequal in size and some of the blocks may be empty. Figure 2 depicts decomposition of a matrix into 16 blocks of equal sizes. These matrix blocks are kept together as one MTBDD. Pointers to the MTBDD nodes representing each block are stored in an array to allow fast access during the sub-MVP operation (line 8, *Compute Process*). The implementation of the MVP operation while the matrix is kept as an MTBDD is explained in Section 4.2. We use $\breve{A}_{pq}$ to refer to the q-th block in the p-th row block of the off-diagonal matrix $\breve{A}$.

The probability vector x is also divided into P blocks of sizes $\{n_0, n_1, \ldots n_{P-1}\}$, where the p-th block is denoted by X_p. The algorithm assumes that, before it commences, an initial approximation for the probability vector x has been stored on disk and that the block X_{P-1} is stored in RAM. In order to schedule the vector x out-of-core, the algorithm requires three arrays of size $n_{\max}$ doubles. The array $\tilde{X}_p$, which is local to the *Compute Process*, is used to accumulate the sub-MVPs (line 8, *Compute Process*). As in Section 2.1, the subscript p for $\tilde{X}_p$ is intended to be intuitive; it does not imply that we have used P such arrays. The other two arrays required are the shared memory buffers, $Xbox_0$ and $Xbox_1$. These are used to read vector blocks from disk (line 8, *Disk-IO Process*). At a certain point in time during the execution, the *Disk-IO Process* is reading a block of iteration vector in one shared buffer, say $Xbox_0$, while the *Compute Process* is consuming a vector block from the other buffer, $Xbox_1$, to accumulate the sub-MVP $\breve{A}_{pq}X_q$. Both processes alternate the value of a local variable t (t is kept hidden in the algorithm for the sake of simplicity) between 0 and 1, in order to switch between the two buffers $Xbox_0$ and $Xbox_1$.

To preserve structure in the symbolic representation, the diagonal elements of the CTMC matrix are stored[1] separately as a vector d. The vector d is also divided into P blocks of sizes $\{n_0, n_1, \ldots n_{P-1}\}$, and is stored on disk. The algorithm uses a shared memory buffer[2] *Dbox* of size $n_{\max}$ short ints to read a block of diagonal vector from disk (line 13, *Disk-IO Process*). The notation for the probability vector described in the paragraph above applies to the diagonal vector: D_p is the p-th block of the diagonal vector d.

The high-level structure of the algorithm, given in Figure 5, is that of a producer-consumer problem. In each execution of its inner `for` loop (lines $6-17$), the *Disk-IO Process* reads the required vector block, X_q, and issues a Signal$(\cdot)$ operation. On receiving this signal, the *Compute Process* issues a return signal and then advances to carry out a *unit of computation*: the sub-MVP $\breve{A}_{pq}X_q$ (line 8, *Compute Process*; see also Figure 2). This activity (lines $5-9$, *Compute Process*) is repeated until all of the blocks in a block row have been read and their products have been accumulated in $\tilde{X}_p$. Once the whole p-th vector block has been updated, the *Compute Process* advances to the next p and signals the *Disk-IO Process*, which receives the signal and writes the new approximation for the p-th block to disk.

We note that, in the p-th phase, all the vector blocks which are required for the computation of the p-th vector block are loaded into RAM from disk such that a diagonal vector block (X_p) follows all the off-diagonal blocks (X_q, $q \neq p$). This ordering is required to perform the convergence test before a block is updated with the new approximation.

[1] Since the number of the distinct values in the diagonal of the matrices considered here is relatively small, n short int pointers to these distinct values are stored instead of n doubles. Also, to save n divisions per iteration, the value *val* of each distinct diagonal entry is stored as $1/val$.

[2] Using one shared block can affect the concurrent execution of the two processes, and hence can worsen the performance. Two shared blocks can be used for the diagonal vector, which may improve the time performance of the algorithm, at a cost of an increase in the memory requirement.

4.2 Computing Sub-MVPs with Symbolic Matrix Storage

The computation of the matrix-vector product (MVP) is the core operation of the iterative symbolic out-of-core algorithm. Its efficiency determines the overall performance of the algorithm. In our approach, the matrix is stored in an offset-labelled MTBDD, as described in Section 3. To implement each sub-MVP, we need to extract the matrix entries for a given matrix block from the MTBDD. As we saw in Section 3, because of its recursive nature, the MTBDD provides a natural decomposition of a matrix into its submatrices. Since an MTBDD is based on binary decisions, descending each level of the data structure splits the matrix into 4 submatrices. Hence, descending l levels, gives a decomposition into $(2^l)^2$ blocks. However, in order to produce an efficient representation, the MTBDD actually encodes a matrix over its *potential state space*, which typically includes many unreachable states. Furthermore, the distribution of these reachable states across the state space is unpredictable. Hence, descending l levels of the MTBDD actually results in blocks of varying and uneven sizes.

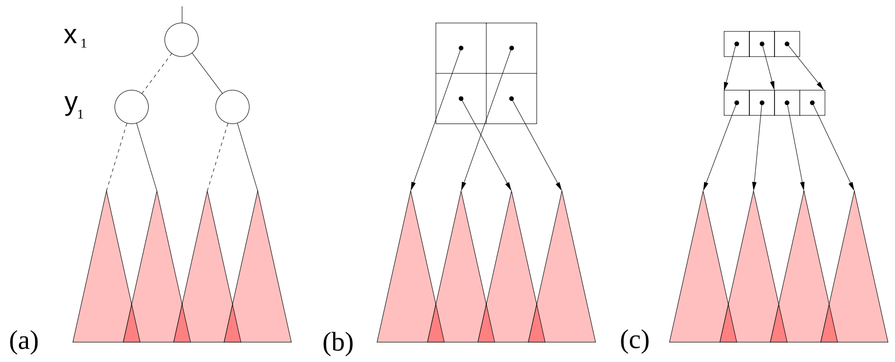


Fig. 6. Matrix block referencing: (a) Original offset-labelled MTBDD (b) $P \times P$ array pointer storage (c) Sparse pointer storage

In our previous implementation of the symbolic out-of-core algorithm, we envisaged that widely varying partition sizes could seriously disrupt the overlap of numerical computation with disk I/O, hindering its performance, and hence we opted to use partitions of equal size. It was therefore necessary to associate each matrix block with several different subtrees of the MTBDD (over a range of levels). Maintaining and referencing this information required additional time.

In this work, we select a value of l, take $P = 2^l$ and use the natural decomposition of the matrix given by the MTBDD. To access each block we simply need to store a pointer to the relevant node of the offset-labelled MTBDD. One possible scheme would be to use a $P \times P$ array of pointers. However, as l grows larger, many of the matrix blocks are empty. Hence, we actually adopt the CSR sparse storage format [33, 34, 25]. These ideas (for the case $l = 1$) are illustrated in Figures 6(a), (b) and (c). The extraction of each row of matrix blocks, as required by the inner loop of the *Compute Process* in Figure 5, is therefore simple and fast.

5 Results

The algorithm presented in Section 4 has been implemented on an UltraSPARC-II 440MHz CPU machine running Solaris with 512MB RAM, and a 6GB local disk. We tested the implementation on three widely used benchmark models: a flexible manufacturing system (FMS) [14], a Kanban system [13] and a cyclic server polling system [23]. These models were generated using PRISM [28], a probabilistic model checker developed at the University of Birmingham. More information about these models, a wide range of other case studies to which PRISM has been applied and the tool itself can be found at the PRISM web site [32]. The work presented in this paper is part of an ongoing effort to improve the range of solution techniques supported by PRISM.

In the next section, we present results of our current implementation of the symbolic out-of-core algorithm and compare these with those of the previous implementation [27]. In Section 5.2, we compare the performance of the symbolic out-of-core algorithm with explicit in-core, explicit out-of-core and symbolic in-core solutions on an identical workstation.

5.1 A Comparison with the Old Implementation

Table 1 summarises results for the Kanban, FMS and polling system case studies. The parameter k in column 2 denotes the number of tokens in the Kanban and FMS models, and the number of stations in the polling system models. Column 3 and 4 list the resulting number of reachable states and off-diagonal non-zero elements in the CTMC matrices. The number of blocks, P, each vector is partitioned into and the resulting total amount of memory used by the current implementation of the out-of-core algorithm are listed in column 5 and 6, respectively. For a large number of vector blocks, 2^l notation is used in column 5, where l is the number of levels in an MTBDD. Accordingly, a matrix is divided into P^2 blocks.

The run times per iteration of the improved symbolic out-of-core solution are recorded in column 8 under the heading "*new*". In order to measure the improvements obtained from this new implementation of the algorithm, we report results for the previous implementation in column 7 under the heading "*old*". A comparison of the two columns confirms a significant improvement in time. In general, we see that the new implementation is approximately twice as fast. All the (out-of-core) solution times are in real time. The symbol '–' in column 7 against a row indicates that the system has not been solved using the old implementation on this workstation. Column 9 ("Iter.") gives the number of iterations the computation took to converge, using the criterion (8) from Section 2. We observe a steady pattern of convergence for all three models. The last column

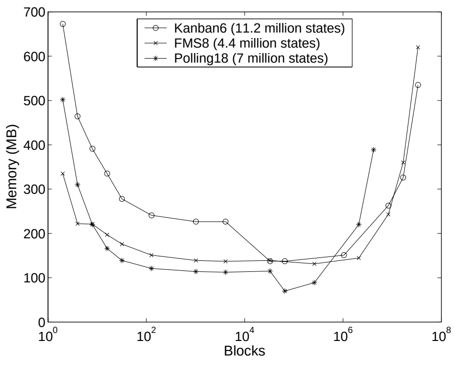
Model	k	States	Off-diagonal	Blocks	RAM	Time (sec/it)		Iter.	MB
		(n)	non-zeros	(P)	(MB)	*old*	*new*		per π
Kanban system	5	2,546,432	24,460,016	2^{10}	133	11.5	4.5	532	20
	6	11,261,376	115,708,992	2^{10}	226	49.4	22.6	717	86
	7	41,644,800	450,455,040	2^{10}	240	215	143	924	317
	8	133,865,325	1,507,898,700	2^{13}	194	1,110	601	1,151	1,004
FMS	8	4,459,455	38,533,968	2^{2}	222	25.41	10.4	1,245	34
	9	11,058,190	99,075,405	2^{23}	220	107.8	35.9	1,416	84
	10	25,397,658	234,523,289	2^{23}	281	380	142	1,591	194
	11	54,682,992	518,030,370	2^{23}	315	1,132	708	1,770	417
	12	111,414,940	1,078,917,632	2^{23}	325	–	1,554	> 50	850
	13	216,427,680	2,136,215,172	2^{23}	392	–	3,428	> 50	1,651
Polling system	18	7,077,888	69,599,232	32	94	21	10.5	302	54
	19	14,942,208	154,402,816	32	124	53	23.8	315	114
	20	31,457,280	340,787,200	32	198	110	52	328	240
	21	66,060,288	748,683,264	32	306	364	177	340	504
	22	138,412,032	1,637,875,712	32	505	795	374	353	1,056

Table 1. Out-of-core numerical solution times for steady-state solution

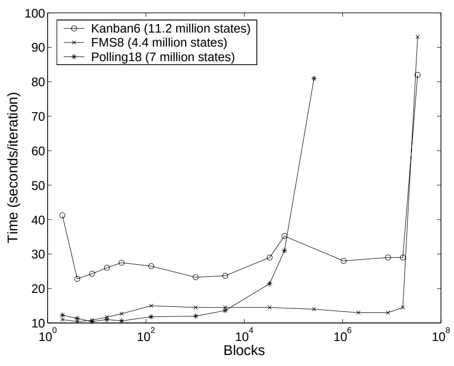
indicates the amount of memory required to store the probability vector (n doubles). Since the vector is stored on disk, this value also corresponds to the file size for the iteration vector.

The first case study in Table 1 is the Kanban system. The largest CTMC solved in this case has 133 million states, which used 194MB of RAM and took 8 days to converge. The matrix was divided into 8192^2 blocks. The second case study is the flexible manufacturing system. The largest model solved in this case is for $k = 11$ with 54 million states. The solution took over 14 days to complete, using 315MB RAM. The largest FMS model we scheduled using our out-of-core solution method is for $k = 13$ with 216 million states. The run times for this model are taken for 50 iterations; we were unable to wait for its convergence, and hence the total number of iterations is not reported in the table. The final case study is the polling system, where the largest model solved is for $k = 22$ which contains 138 million states. The out-of-core solution for this system used 505MB of RAM and took 1.5 days to converge.

Generally, the memory required by our out-of-core solution can be decreased by increasing the number of blocks. This is due to the fact that increasing the number of vector blocks typically causes a reduction in the size of the largest vector block to be kept in RAM, i.e. reduces $n_{\max}$. Figure 7(a) illustrates this for three CTMCs, one from each model. The plots display the total amount of memory used against the number of vector blocks. Consider the plot for the Kanban system ($k = 6$). The memory required for the case $P = 2$ is above 650MB. Increasing the number of blocks reduces the memory requirements to the minimum: nearly 140MB. The same properties are evident for the FMS ($k = 8$) and polling system ($k = 18$) CTMCs. For large numbers of blocks (i.e. the rightmost portions of the plots), we note an increase in the amount



(a) Memory requirements vs. blocks



(b) Time per iteration vs. blocks

Fig. 7. Memory and time usage of the symbolic out-of-core solution

of memory. This is because the memory overhead required to store information about the blocks of the MTBDD dominates the overall memory in these cases.

In Figure 7(b), we analyse the time per iteration for the same three CTMCs, plotted against the number of vector blocks. We explain the plot for the Kanban system ($k = 6$). Initially, for $P = 2$, the memory required (see Figure 7(a)) for solution is more than the available RAM. This causes thrashing and results in a high solution time. An increase in the number of blocks removes this problem and explains the initial downward jump in the plot. From this point on, however, the times vary. The explanation for this is as follows. Our decomposition of the matrix into blocks can result in a partitioning such that there is a significant difference between the maximum and minimum partition sizes. This can affect the overlap of computation and disk I/O, effectively increasing the solution time. The sizes of the partitions are generally unpredictable, being determined both by the sparsity pattern of the matrix and by the encoding of the matrix into the MTBDD. Finally, we note that the end of the plot shows an increase in the solution time. This is due to the overhead of manipulating a large number of blocks of the matrix and the increased memory requirements that this imposes, as is partially evident from Figure 7(a).

Similar patterns can be observed for the other two plots in Figure7(b). Likewise, we have found that varying the value of k within each case study also results in similar patterns. This can, in fact, be useful for predicting good choices of partition sizes for larger values of k. A useful direction for future work would be to investigate more fully what constitutes a good choice of P.

It is evident from Table 1 and Figure 7(b) that the FMS system has the highest solution times per iteration, given an equal number of states. The lowest solution times among the three reported models are attributed to the polling system. The FMS system is the least structured of the three models which equates to a large MTBDD to store it; the larger the MTBDD, the more time is required to perform its traversal. The polling system, on the other hand, is very structured and therefore results in a smaller MTBDD.

5.2 A Comparison of In-Core and Out-of-Core Solutions

In this section, we compare in-core and out-of-core versions of both symbolic and explicit implementations. The results are summarised in Table 2. They were collected on the same machine – an UltraSPARC-II 440MHz CPU machine with 512MB RAM and a 6GB local disk. All reported run times are in real time.

The first three columns of Table 2 are identical to Table 1 except that this time we have reported a larger range of the parameter k for each model. Columns 4–5 in the table list the results for explicit implementations: both standard "in-core", where the matrix and the vector are kept in RAM; and "out-of-core", as in [26], where they are stored on disk. In both cases, the matrix is stored using the *Compact MSR* sparse matrix storage scheme [26] and the vector is stored as an array.

Model	k	States	Time (seconds per iteration)			
			Explicit		Symbolic	
			in-core	out-of-core	in-core	out-of-core
Kanban system	4	454,475	0.4	1.2	0.5	1.3
	5	2,546,432	2.3	7.0	3.1	4.5
	6	11,261,376	–	69	15.6	22.6
	7	41,644,800	–	308	–	143
	8	133,865,325	–	–	–	601
FMS	6	537,768	0.5	2.2	1.0	1.3
	7	1,639,440	1.6	7	3.0	3.2
	8	4,459,455	3.9	21	8.9	10.4
	9	11,058,190	–	57	39.6	35.9
	10	25,397,658	–	178	149	142
	11	54,682,992	–	–	–	708
	12	111,414,940	–	–	–	1,554
	13	216 427 680	–	–	–	3,428
Polling system	15	737,280	0.5	3.0	0.8	0.8
	16	1,572,864	1.1	7.2	1.7	2.0
	17	3,342,336	2.58	15.2	3.9	4.6
	18	7,077,888	5.80	34.3	8.3	10.5
	19	14,942,208	–	68.6	20.3	23.8
	20	31,457,280	–	132	155	52
	21	66,060,288	–	–	–	177
	22	138,412,032	–	–	–	374

Table 2. Comparing solution times for in-core and out-of-core solutions

Columns 6–7 in Table 2 report results for symbolic implementations: both "in-core", where the offset-labelled MTBDD for the matrix and the array for the vector are kept in RAM, and "out-of-core", as described in this paper. The run times for the former were obtained using the existing in-core implementation in the tool PRISM. The times for the latter are identical to those reported in Table 1. The relative performances of these two symbolic implementations are plotted against the number of states in Figure 8. The smaller the value of this ratio, the better the performance of the out-of-core approach. Ideally, we would like this value to be as close to 1 as possible, i.e. minimising the additional time overhead incurred due to the out-of-core scheduling of the vector. We see from Table 2 and Figure 8 that, in general, we achieve this aim. In fact, when the memory usage of the in-core technique approaches or exceeds the total RAM available, this ratio is actually less than 1 due to thrashing. For the case where the in-core method is incapable of scheduling a CTMC model, we consider the performance ratio to be zero (see Figure 8).

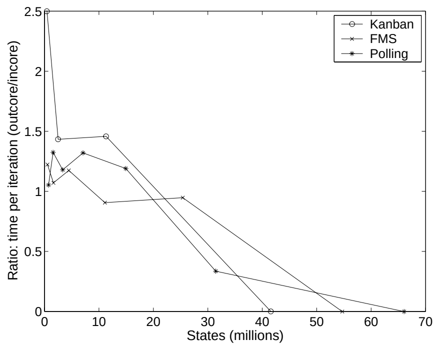


Fig. 8. Symbolic out-of-core vs. symbolic in-core

We note in Table 2 that the in-core solution of column 4 provides the fastest run-times. However, pursuing this approach, the largest model solvable on the 512MB workstation is the polling system ($k = 18$) with approximately 7 million states. The in-core symbolic solution of column 6 can solve larger models because, in this case, the matrix is stored symbolically. The largest model solvable with this symbolic in-core approach is the polling system ($k = 20$) with 31 million states. The out-of-core storage of matrix (explicit) and vector can solve even larger models. This is reported in column 5, and the largest model reported in this case is the Kanban system ($k = 7$) with 41 million states, although, using this approach, solution of even larger models is possible.

We observe that the results for explicit solutions are quite consistent with all three example models. However, the performance of implicit (symbolic) solutions, both in-core and out-of-core, depends on a particular system. This is because the symbolic methods rely on structure in a model. We conclude with our observation of Table 2 that the symbolic out-of-core solution provides the best overall results for the examples considered.

6 Conclusion

In this paper, we have presented an improved implementation of our symbolic out-of-core algorithm and given a detailed analysis of its performance. The implementation has been tested on three benchmark models. Generally, we observe a speedup of approximately a factor of two over our earlier approach. We report solution of models as large as with 138 million states on a workstation with 512MB RAM. Even larger systems with up to 216 million states have been shown to be solvable using this approach. This is demonstrated by scheduling these models for steady-state solution on the workstation. Since the equivalent implicit approaches require an iteration vector of size proportional to the state space, the largest model these techniques can solve on equivalent hardware is of size 64 million states.

The idea of symbolic out-of-core solution is a promising one, and is equally applicable to other symbolic methods such as matrix diagrams or Kronecker methods. Although it has been demonstrated in this paper that very large models can be solved on a modern workstation using our symbolic out-of-core approach, the solution process for large models is quite slow. In future we will extend this approach by employing parallelisation. We also intend to generalise these techniques to other numerical computation problems, such as transient analysis of CTMCs and analysis of DTMCs and MDPs.

7 Acknowledgement

We are grateful to Gethin Norman for his assistance in developing the case studies which we have used for experimental results in this paper.

References

1. S. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In *Proc. PNPM'97*, pages 112–121. IEEE Computer Society Press, 1997.
2. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications. In *Proc. ICCAD'93*, pages 188–191, Santa Clara, 1993.
3. C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic Model Checking for Probabilistic Processes. In *Proc. ICALP'97*, pages 430–440, April 1997. Available as Volume 1256 of *LNCS*.
4. R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics, 1994.
5. A. Bell and B. R. Haverkort. Serial and Parallel Out-of-Core Solution of Linear Systems arising from Generalised Stochastic Petri Nets. In *Proc. High Performance Computing 2001*, Seattle, USA, April 2001.

6. Marius Bozga and Oded Maler. On the Representation of Probabilities over Structured Domains. In N. Halbwachs and D. Peled, editors, *Proceedings of CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 261–273. Springer-Verlag, July 1999.
7. P. Buchholz. Structured analysis approaches for large Markov chains. *SIAM Journal on Matrix Analysis and Applications*, 31(4), 1999.
8. P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Kronecker Operations and Sparse Matrices with Applications to the Solution of Markov Models. ICASE Report 97-66, Institute for Computer Applications in Science and Engineering, December 1997.
9. P. Buchholz, M. Fischer, and P. Kemper. Distributed Steady State Analysis Using Kronecker Algebra. In *Proc. NSMC'99*, pages 76–95, 1999.
10. P. Buchholz and P. Kemper. Compact Representations of Probability Distributions in the Analysis of Superposed GSPNs. In Reinhard German and Boudewijn Haverkort, editors, *Proc. PNPM'01*, pages 81–90, September 2001.
11. S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on SIMD massively parallel GSPN analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 266–283. Springer-Verlag, 1994.
12. G. Ciardo and A. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proc. PNPM'99*, Zaragoza, 1999.
13. G. Ciardo and M. Tilgner. On the use of Kronecker Operators for the Solution of Generalized Stochastic Petri Nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.
14. G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.
15. Gianfranco Ciardo. Distributed and structured analysis approaches to study large and complex systems. *Lecture Notes in Computer Science*, 2090:344–374, 2001.
16. E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Effificient Data Structure for Matrix Representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, May 1993.
17. D. D. Deavours and W. H. Sanders. An Efficient Disk-based Tool for Solving Very Large Markov Models. In Raymond Marie et al., editor, *Proc. TOOLS'97*, volume 1245 of *LNCS*, pages 58–71. Springer-Verlag, 1997.
18. D. D. Deavours and W. H. Sanders. An Efficient Disk-based Tool for Solving Large Markov Models. *Performance Evaluation*, 33(1):67–84, 1998.
19. D. D. Deavours and W. H. Sanders. "On-the-fly" Solution Techniques for Stochastic Petri Nets and Extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.
20. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications. Clarendon Press Oxford, (with corrections)1997.
21. G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian Analysis of Large Finite State Machines. *IEEE Transactions on CAD*, 15(12):1479–1493, 1996.
22. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. Numerical Solutions of Markov Chains (NSMC'99)*, Zaragoza, 1999.
23. O. Ibe and K. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.
24. William J. Knottenbelt and Peter G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In *Proc. Numerical Solution of Markov Chains (NSMC'99)*, Prensas Univerversitarias de Zaragoza, 1999.

25. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cumming Publishing Company, 1994.
26. M. Kwiatkowska and R. Mehmood. Out-of-Core Solution of Large Linear Systems of Equations arising from Stochastic Modelling. In *Proc. PAPM-PROBMIV'02*, July 2002. Available as Volume 2399 of *LNCS*.
27. M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A Symbolic Out-of-Core Solution Method for Markov Models. In *Proc. Parallel and Distributed Model Checking (PDMC'02)*, August 2002. Appeared in Volume 68, issue 4 of ENTCS (http://www.elsevier.nl/locate/entcs).
28. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. TOOLS'02*, volume 2324 of *LNCS*, April 2002.
29. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *Proc. TACAS 2002*, April 2002. Available as Volume 2280 of *LNCS*.
30. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, August 2002.
31. B. Plateau. On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms. In *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 147–153, 1985.
32. PRISM Web Page. http://www.cs.bham.ac.uk/~dxp/prism/.
33. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, NASA Ames Research Center, Moffett Field, CA, 1990.
34. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, 1996.
35. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
36. Sivan Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.