A Symbolic Out-of-Core Solution Method for Markov Models ⋆

Marta Kwiatkowska, Rashid Mehmood, Gethin Norman and David Parker

School of Computer Science, University of Birmingham, Birmingham B15 2TT, United Kingdom

Abstract

Despite considerable effort, the state-space explosion problem remains an issue in the analysis of Markov models. Given structure, symbolic representations can result in very compact encoding of the models. However, a major obstacle for symbolic methods is the need to store the probability vector(s) explicitly in main memory. In this paper, we present a novel algorithm which relaxes these memory limitations by storing the probability vector on disk. The algorithm has been implemented using an MTBDD-based data structure to store the matrix and an array to store the vector. We report on experimental results for two benchmark models, a Kanban manufacturing system and a flexible manufacturing system, with models as large as 133 million states.

1 Introduction

Discrete-state Markovian models are widely employed for the analysis of communication networks and computer systems. It is often convenient to model such systems as Continuous Time Markov Chains (CTMCs), provided probability distributions are assumed to be exponential. A CTMC may be represented by a set of states and a transition rate matrix Q containing state transition rates as coefficients. A CTMC can be analysed using probabilistic model checking. Required or desired performance properties are specified as formulas in the temporal logic CSL and then automatically verified using the appropriate model checking algorithms. A core component of these algorithms is the computation of the steady-state probabilities of the CTMC. This is reducible to the classical problem of solving a system of linear equations of the form $Ax = b$ where $b = 0$. A range of solution techniques exist to combat the

⋆ Supported in part by the EPSRC grant GR/N22960.

[1] Email: rxm@cs.bham.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

so-called state-space explosion (also known as largeness) problem in this area. These include symbolic techniques [8,13,15,25,14], *on-the-fly* methods [11] and Kronecker methods [22]. While some new developments such as Matrix Diagrams (MDs) [5,4,20] and hybrid MTBDD methods [19] allow for a very compact and time efficient encoding of CTMC matrices, they are obstructed by explicit storage of the probability vector(s) needed during the numerical solution phase.

An established direction of research in the area of Markov modelling has concerned numerical solution techniques which store the CTMC matrix explicitly, using a sparse storage format. These are the most generally applicable techniques, fast though not as compact as the symbolic methods. Improvements have been obtained through using disks (so called out-of-core techniques [2]) to store the CTMC matrix [10] and parallelising the disk-based numerical solutions [16,2]. However, the in-core memory requirements for these methods are also dominated by the storage of a vector with size proportional to the number of states in the model.

The authors have presented, in [17], an out-of-core algorithm which relaxes the above mentioned memory limitations on the explicit sparse methods. In this paper, we introduce a novel algorithm which relaxes the same memory limitations for symbolic methods; we call the algorithm a *symbolic out-of-core algorithm*.

The philosophy behind our symbolic out-of-core algorithm is to store the CTMC matrix using the MTBDD-based data structure of [19] and to store the probability vector explicitly on disk. The algorithm divides the probability vector into a certain number of blocks for this purpose. Since the matrix is stored in an MTBDD-based data structure, blocks can be extracted as needed in the calculation. It reads into main memory a block of the probability vector, performs the required calculation using a matrix block extracted from the MTBDD-based data structure and writes back updated elements of the probability vector onto the disk. To obtain performance from the algorithm, the work is divided between two concurrent processes; one performs the computation while the other schedules the disk reading and writing. The memory requirement of the algorithm is dependent on the number of blocks; the higher the number of blocks the probability vector is divided into, the less memory is required by the algorithm at a cost of increased run time. We give experimental results from the implementation of the symbolic out-of-core algorithm applied to a Kanban manufacturing system [6] and a Flexible Manufacturing System (FMS) [7].

[2] Algorithms that are designed to achieve high performance when their data structures are stored on disk are known as *out-of-core algorithms*; Toledo presents a survey of such algorithms in numerical linear algebra, see [27].

1.1 The Tool PRISM

All the CTMC matrices used in this paper were generated by the PRISM tool [18]. PRISM, a probabilistic model checker being developed at the University of Birmingham, is a tool for analysing probabilistic systems. It supports three models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. For the numerical solution phase, it provides three engines: one using pure MTBDDs, one based on in-core sparse iterative methods, and a third which is a hybrid of the first two. Several case studies have been analysed [23]. This paper is part of an effort to improve the range of solution methods provided by PRISM.

2 Numerical Considerations

We focus in this paper on the numerical solution of continuous time Markov chains. The task of solving a CTMC to obtain the steady-state probabilities vector can be mathematically written as:

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

where $Q \in \mathbb{R}^{n\times n}$ is the infinitesimal generator matrix of the CTMC, and $\pi \in \mathbb{R}^n$ is the steady state probability vector. The matrices Q are usually very sparse; further details about the properties of these matrices can be found in [26]. The 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.

The numerical solution methods for linear systems of the form $Ax = b$ are broadly classified into direct methods and iterative methods. For large systems, direct methods become impractical due to the phenomenon of fill-in, caused by the generation of new entries during the factorisation phase. Iterative methods generate a sequence of approximations that only converges in the limit to the solution. Beginning with a given approximate solution, these methods modify the components of the approximation in each iteration, until a required accuracy is achieved. See [12] for further information on direct methods and [24,26] for iterative methods.

We concentrate here on stationary iterative methods. In each iteration of the Jacobi method, for example, we calculate:

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

for $0 \le i < n$, where a_{ij} denotes the element in row i and column j of matrix A. We note in equation (2) that the new approximation of the solution vector

$(x_i^{(k)})$ is calculated using only the old approximation of the solution $(x_j^{(k-1)})$. The Gauss-Seidel (GS) iterative method, which in practice converges faster than the Jacobi method, uses the most recently available approximation of the solution:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i \; - \; \sum_{j<i} x_j^{(k)} a_{ij} \; - \; \sum_{j>i} x_j^{(k-1)} a_{ij} \right), \tag{3}$$

for $0 \leq i < n$.

In the Jacobi method, the order in which entries of A are accessed within a single iteration is unimportant. For Gauss-Seidel, access to individual columns is required. For these reasons, symbolic implementations of iterative methods based on MTBDDs are better suited to Jacobi than Gauss-Seidel. Parker [21] resolves this problem by introducing the 'Pseudo Gauss-Seidel' method, a compromise between Jacobi and Gauss-Seidel. Assuming that the matrix A is split into $B \times B$ blocks of size $n/B \times n/B$, Pseudo Gauss-Seidel can be described as:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<N_i} a_{ij} \cdot x_j^{(k)} - \sum_{j \geq N_i \wedge j \neq i} a_{ij} \cdot x_j^{(k-1)} \right)$$

where $N_i = \lfloor i/(n/B) \rfloor \cdot n/B$. Because access to the entries of A is on a block-by-block basis, this method is also well suited to MTBDD-based methods. Since each iteration uses *some* elements of the most recent approximation, Pseudo Gauss-Seidel generally converges faster than Jacobi.

The convergence characteristics of Pseudo Gauss-Seidel for the computation of steady-state probabilities are similar to those of Jacobi and Gauss-Seidel, as presented for example in [26]. This is because, like the other two methods, Pseudo-Gauss Seidel can be shown to be based on a *regular splitting*. See [21] for more details.

In all the experiments presented in this paper, we chose the relative error criterion:

$$\max_i \left(\frac{\mid x_i^{(k)} - x_i^{(k-1)} \mid}{\mid x_i^{(k)} \mid} \right) \; < \; 10^{-7}. \tag{4}$$

3 Symbolic CTMC Storage

MTBDDs (Multi-Terminal Binary Decision Diagrams) [8,1] 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. By encoding their indices as Boolean variables, real-valued vectors and matrices can be represented as MTBDDs. It has been shown in [8,1,13,15] how basic operations such as matrix addition and matrix-vector multiplication can be performed with MTBDDs and how this can be used to implement iterative numerical solution techniques such as the Power and Jacobi methods.

The most important advantage of the MTBDD data structure is that it can provide extremely compact storage for very large, structured matrices, such as those derived from high-level descriptions of CTMCs. Unfortunately, the performance of MTBDD-based numerical computation has often been found to be poor, especially in comparison to traditional, explicit implementations based on sparse matrices and arrays. The drawback of sparse matrices, however, is that they can be expensive in terms of memory.

In [18], a hybrid approach to numerical solution is presented, representing the matrix as an MTBDD and the solution vector as an array. This is 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 indices into the solution vector. It was found that this hybrid approach retained the compact storage advantages of MTBDDs and could almost match the solution speed of sparse matrices.

4 Related Work

We use the term "implicit methods" in this paper for the numerical solution methods which use some kind of symbolic data structure for the storage of a CTMC matrix. The term "explicit" will be used to denote the numerical solution methods which store the CTMC matrix using a sparse data structure.

Implicit methods offer a compact representation of structured Markov models and enable the fastest solutions for very large models with a certain structure. Several developments in the area of symbolic methods for the analysis of Markovian models have occurred recently. We note the hybrid symbolic approach of Kwiatkowska et al. [19] whose solution method is based on Jacobi iteration. An alternative is the structural decomposition approach of Ciardo and Miner [5,4,20], which results in a flexible data structure called Matrix Diagrams (MDs) that can be used with the Gauss-Seidel iterative method for the analysis of a wide range of models. However, as these researchers have stated, symbolic methods are hindered by the memory requirement for the storage of the probability vector(s).

In Table 1, we summarise the main representative approaches used for overcoming the state space explosion problem when analysing stochastic models. We concentrate on the data structures used to store the matrix and vector, and whether they are stored in- or out-of-core.

The MTBDD-based methods use MTBDDs to store both the matrix as well as the vector (in-core). This approach becomes impractical for large models because of inefficient MTBDD representation of the probability vector, despite the fact that the MTBDD representation of the CTMC matrix can be very compact (using less memory than the vector). The hybrid approach of [19] combines MTBDDs and explicit methods (§3). The matrix is stored using an MTBDD, labelled with offsets, and the vector is stored in-core, as an array. Its limitation is the explicit storage of the probability vector(s). The next two approaches (Kronecker and PDGs) are based on a compact representation of the CTMC matrix using Kronecker expressions. The first of these offers a compact representation of the CTMC matrix but requires explicit storage of the vector. The second uses the Probabilistic Decision Graph data structure to provide a symbolic (hence compact) storage of the probability vector. In practice, it has been shown that this is not very efficient. The Matrix Diagrams of Ciardo and Miner also suffer from the explicit storage of the probability vector. The *symbolic out-of-core* approach introduced in this paper is listed next, where the matrix is stored in-core using an offset-labelled MTBDD [19] and the vector is stored on disk. This approach does not have the memory limitations stated for all other implicit methods in Table 1.

The first explicit out-of-core method listed in Table 1 was introduced by Deavours and Sanders [9,10], where the vector is stored explicitly in RAM as an array and a disk is used to store the matrix explicitly. In [17] this method was further extended by storing the vector as well as the matrix on disk. This extension relaxes the memory limitations on out-of-core methods caused by the need to store the probability vector in RAM.

Method	Matrix (Q)		Vector (π)	
	Data structure	In-core/Out-of-core	Data structure	In-core/Out-of-core
Implicit Methods				
MTBDDs [13,15,14]	MTBDD	In-core	MTBDD	In-core
Hybrid [19,21]	Offset-labelled MTBDD	In-core	Array	In-core
Kronecker [22]	Kronecker Expression	In-core	Array	In-core
PDGs [3]	Kronecker Expression	In-core	PDG	In-core
Matrix Diagrams [5,4,20]	Matrix Diagram	In-core	Array	In-core
This paper	Offset-labelled MTBDD	In-core	Array	Out-of-core
Explicit Methods				
Out-of-core [9,10]	Sparse matrix	Out-of-core	Array	In-core
Out-of-core [17]	Sparse matrix	Out-of-core	Array	Out-of-core

Table 1
Various storage schemes for CTMC analysis

This paper reports on a new approach combining ideas from symbolic [19] and out-of-core [17] methods. This new approach, which we call the *symbolic out-of-core* method, uses the offset-labelled MTBDD data structure of [19] to store the CTMC matrix in-core, while the probability vector is kept explicitly on a disk. As the name implies, the method is a composite of out-of-core and symbolic techniques. The aim of this approach is to eliminate the bottleneck imposed on symbolic methods, caused by the need to explicitly store the probability vector in main memory.

The main difference between the symbolic out-of-core method introduced in this paper and our earlier out-of-core approach of [17] lies in the way the CTMC matrix is stored. In [17], the matrix is stored explicitly on disk, as opposed to symbolically in this paper; the probability vector is stored out-of-core explicitly for both approaches. The advantage of the symbolic out-of-core approach over the out-of-core algorithm of [17] is reduced disk I/O at a cost of increased CPU usage. However, the limitations on symbolic methods caused by the need for matrices to have structure apply to the symbolic out-of-core solution, whereas the method of [17] has no such restrictions.

5 A Symbolic Out-of-Core Solution with MTBDDs

The philosophy behind the symbolic out-of-core solution introduced here is to keep the matrix in-core, in an appropriate symbolic data structure, and to store the probability vector on disk. The iteration vector can be divided into any number of blocks and these blocks can be stored on a disk. During the iterative computation phase, these vector blocks can be fetched from disk, one after another, into main memory to perform the numerical computation. We have used offset-labelled MTBDDs [19,21] for CTMC storage, while the iteration vector for numerical computation has been kept on disk as an array.

5.1 The Algorithm

The symbolic out-of-core algorithm, given in Figure 1, comprises two concurrent processes: the *Disk-IO process* and the *Compute process*. We have implemented this algorithm on UNIX and Linux using two separate processes communicating via shared memory and synchronising using semaphores. The algorithm achieves communication between the two processes by using the shared memory blocks *Dbox* and Πbox_x. The processes synchronise by calling the functions Wait($\cdot$) and Signal($\cdot$).

The algorithm assumes that the CTMC matrix to be solved is stored in-core, using the offset-labelled MTBDD data structure [19]. The matrix is virtually divided into $B \times B$ square blocks of dimension $n/B \times n/B$, where some of the blocks might be empty. Figure 2 shows the division of a matrix resulting from the Flexible Manufacturing System (FMS) model of [7] into 4×4 blocks. What we mean here by “virtually” is described later in this

Integer constant: B (*number of blocks*)
Semaphores: S_1, S_2: occupied
Shared variable: *Dbox (to read diagonal blocks into RAM)*
Shared variables: Πbox_0, Πbox_1 *(to read solution vector π blocks into RAM)*

Disk-IO process

```
1.  Local variable: h, i, j, k
2.  k ← B − 1
3.  while not converged
4.     for i = 0 to B − 1
5.        if i = 0 then j ← B − 1
6.        else  j ← i − 1
7.        for h = 0 to B − 1
8.           if h ≠ 0
9.              if not an empty block
10.                read Π_j from disk
11.          Signal(S_1)
12.          Wait(S_2)
13.          if h = 0
14.             read D_i into Dbox
15.             write Π_k to disk
16.          if j = 0 then j ← B − 1
17.          else  j ← j − 1
18.       k ← k + 1  mod B
```

Compute process

```
1.  Local variable: i, j
2.  while not converged
3.     for i = 0 to B − 1
4.        if i = 0 then j ← B − 1
5.        else  j ← i − 1
6.        for 0 to B − 1
7.           Wait(S_1)
8.           Signal(S_2)
9.           if not an empty block
10.             Accumulate Q_ij Π_j
11.          if j = 0 then j ← B − 1
12.          else  j ← j − 1
13.       Update Π_i using D_i
14.       check for convergence
```

Fig. 1. The symbolic out-of-core Pseudo Gauss-Seidel iterative algorithm

section, in the context of matrix-vector block multiplication. The algorithm assumes $n \bmod B = 0$.

The probability vector π is also divided into B blocks, each with n/B elements. In order to avoid confusion between the j-th element of the probability vector π and its j-th block, we use π_j to indicate the j-th element of the vector π and Π_j to indicate the j-th block of the vector π; Π_{ij} stands for the j-th element of the block Π_i. The algorithm also assumes that the initial approximation for all the blocks of the probability vector π is stored on disk except for the last block Π_{B-1}.

To preserve structure in the symbolic representation, the diagonal elements of the CTMC matrix are stored[3] separately as a vector d. The vector d is also divided into B blocks with n/B elements each, and is stored on disk. A diagonal block is fetched from disk (line 14 in disk-IO process) into *Dbox* and is overwritten by another diagonal block after use. The notation for the

[3] Since the number of distinct values in the diagonal of the matrices considered here is relatively small, n short pointers to these distinct values are stored instead of doubles.

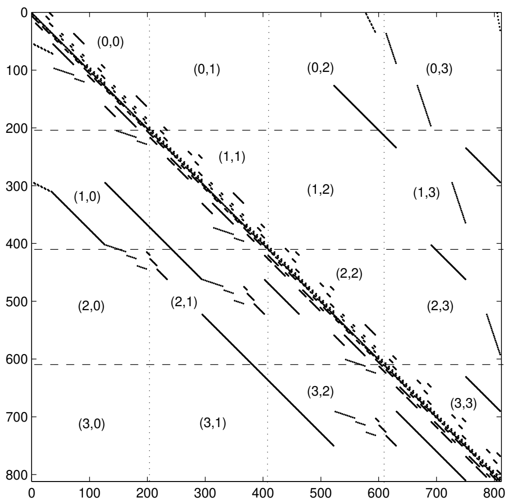


Fig. 2. Division of the CTMC matrix of an FMS model into 4×4 square blocks

probability vector described in the above paragraph applies to the diagonal vector; d_j stands for the j-th element of the diagonal vector and D_k is the k-th block of the diagonal vector.

5.1.1 Disk-IO Process.

The Disk-IO process is responsible for all the file I/O work. It reads the diagonal and the probability vector blocks from disk into RAM, and writes the new approximations of the probability vector onto disk. If required, the Disk-IO process reads a vector block in each iteration of the inner loop (lines $7-17$). The index of a particular block to be read is determined using the lines $5-6$ along with the lines $16-17$. The process uses two shared memory arrays – Πbox_0 and Πbox_1 – to read the blocks from disk into RAM. At one point in time, one shared memory array (Πbox_x) is used by the Compute process for the multiplication of a matrix block and a vector block ($Q_{ij}\Pi_j$), while the other shared memory array (Πbox_x) is used by the Disk-IO process to read the next required vector block from disk. The variable x alternates between 0 and 1 locally within each process to determine the shared memory block to be used in a particular iteration of the inner loop. For the sake of simplicity, we have omitted from the algorithm in Figure 1 information regarding the indexing of shared memory segments.

In each of the first iterations ($h = 0$) of the inner loop (lines $7 - 17$), the new approximation of a vector block is written to disk instead of a read operation for a vector block. The index for the block to be written to disk is determined by the variable k. Another shared memory array ($Dbox$) of n/B *short* integers is used to read a block of the diagonal vector into RAM. The diagonal block is read in each of the first iterations ($h = 0$) of the inner loop. On the other hand, the Compute process uses the shared memory block ($Dbox$) during each of the last iterations of the inner loop. Using two shared blocks for the diagonal vector may possibly improve the time performance of the algorithm with an increase in the memory requirement.

The high-level structure of the algorithm is that of a producer-consumer problem. The Disk-IO process begins by issuing the Signal($\cdot$) operation after reading a vector block from disk. Since both semaphores – S_1 and S_2 – are occupied innitially, the Compute process cannot advance until the Signal($\cdot$) operation is carried out by the Disk-IO process.

5.1.2 Compute Process.

This process is responsible for all the numerical computations involved in the steady-state solution of a CTMC model. The numerical iterative method we employed in this algorithm is the Pseudo Gauss-Seidel method (see §2). Since the vector π is read into RAM one block after another, we accumulate the products $Q_{ij}\Pi_j$ in an array of *doubles* of size n/B for all $0 \leq j < B$. These products are accumulated by line 10 of the Compute process in a local array. The index of the matrix and vector blocks used in the products is determined using the lines $3 - 4$ along with the lines $11 - 12$. The next approximation for the block Π_i is calculated by dividing the accumulated products by the diagonal vector block (D_i).

The Compute process starts with a Wait($\cdot$) operation. After a signal operation from the Disk-IO process, it issues a signal on S_2 and proceeds to the matrix-vector block multiplication. This process is repeated until all the required matrix-vector block products have been accumulated for the calculation of a vector block. The Compute process then updates the vector block and tests for convergence. The extraction of matrix entries from the MTBDDs, as required for this process, is described in the next section.

5.1.3 Matrix-Vector Block Multiplication using MTBDDs.

It has been mentioned in §2 that an MTBDD is a rooted, directed acyclic graph. The entries of a matrix stored in an MTBDD can be extracted by traversing the data structure from the root node. Since we store the whole CTMC matrix in a single offset-labelled MTBDD data structure (in-core), the naïve approach to the matrix-vector block multiplication ($Q_{ij}\Pi_j$), which is required in each iteration of the inner loop (line 10 of the Compute process), is to traverse the whole data structure from the root node. The problem with this approach is that, during the traversal, only those matrix entries which are required for this particular block multiplication are used, while the remaining matrix elements have to be ignored.

Therefore, as in our earlier out-of-core algorithm [17], we would like to store the equally sized square matrix blocks separately, as shown in Figure 2. However, if we took this approach, the structure of the matrix (which is exploited by the MTBDD data structure and results in a compact representation) would be lost. Hence, we decide to keep the complete matrix in one data structure and, to make our symbolic out-of-core algorithm time efficient, we store pointers to the edges in the MTBDD taken for each matrix-vector block multiplication. The extra memory required to keep this additional information varies with the number of blocks and is independent of the number of states. For up to 64×64 matrix blocks, this memory does not exceed 2 MB.

5.2 Results

We have implemented our symbolic out-of-core algorithm on two workstations: an AMD Athlon[tm] 1600MHz dual processor machine running Linux with 900MB RAM (machine1), and an UltraSPARC-II 440MHz CPU machine running UNIX with 512MB RAM (machine2). The implementation has been tested on two large benchmark models available in the literature, a Kanban System [6] and a Flexible Manufacturing System (FMS) [7].

Table 2 presents performance measures of the algorithm for FMS and Kanban CTMC matrices, implemented and executed on machine1. The parameter l in column 2 of the table denotes the number of tokens in the FMS or Kanban system; column 3 and 4 list the resulting number of reachable states and off-diagonal nonzero elements in the CTMC matrices. The number of blocks each matrix is partitioned into and the resulting total amount of memory used are listed in column 5 and 6 respectively. The figure for RAM listed in column 6 includes the memory required for all the vector blocks, for keeping matrix in offset-labelled MTBDD format, memory for file buffering and for keeping additional block information.

We note that increasing the number of blocks reduces the size of the shared memory segments and hence the memory requirement of the solution process, while on the other hand increasing the solution time. The number of blocks for a CTMC matrix is chosen based on the size of the CTMC and the assumption that $n \bmod B = 0$. For small CTMCs, we divide the vector into 4 to 8 blocks. As the size of the CTMC increases, we divide into a larger number of blocks

Model	l	States (n)	Off-diagonal non-zeros (a)	Blocks	RAM (MB)	Time (sec/it)		Iter.	MB per π
						incore	*outcore*		
Kanban	5	2,546,432	24,460,016	4	21	0.81	2.39	565	20
	6	11,261,376	115,708,992	4	91	3.92	12.7	767	86
	7	41,644,800	450,455,040	4	359	292	57.4	1003	317
	8	133,865,325	1,507,898,700	55	79	–	1110	1044	1004
FMS	8	4,459,455	38,533,968	5	29	1.95	5.98	1255	34
	9	11,058,190	99,075,405	5	75	8.23	26.7	1424	84
	10	25,397,658	234,523,289	6	138	23.5	73.3	1605	194
	11	54,682,992	518,030,370	8	230	832	257	1784	417
	12	111,414,940	1,078,917,632	17	224	–	6734	> 100	850

Table 2

Numerical solution times for the FMS and the Kanban models on machine1

Model	l	States (n)	Off-diagonal non-zeros (a)	Blocks	RAM (MB)	Time (sec/it)		Iter.	MB per π
						incore	*outcore*		
Kanban	5	2,546,432	24,460,016	4	21	3.07	11.5	565	20
	6	11,261,376	115,708,992	4	91	15.2	49.4	767	86
	7	41,644,800	450,455,040	8	180	772	215	1003	317
FMS	8	4,459,455	38,533,968	5	29	8.61	25.41	1255	34
	9	11,058,190	99,075,405	5	75	39.9	107.8	1424	84
	10	25,397,658	234,523,289	18	46	580	380	1594	194

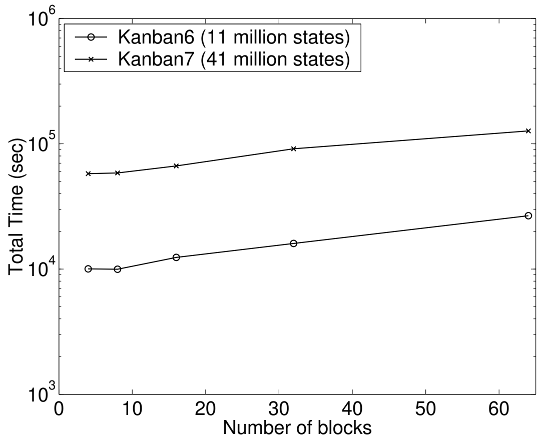
Table 3
Numerical solution times for the FMS and the Kanban models on machine2

such that each block can fit within the available RAM. For example, Kanban ($l = 8$) was solved using both 25 and 55 blocks. The solution time in both cases was almost the same, and we reported the result with 55 blocks in the table. For large models, an increase in the number of blocks may lead to a decrease in the solution time due to a caching effect. Figure 3(b) demonstrates such behaviour for FMS ($l = 11$).

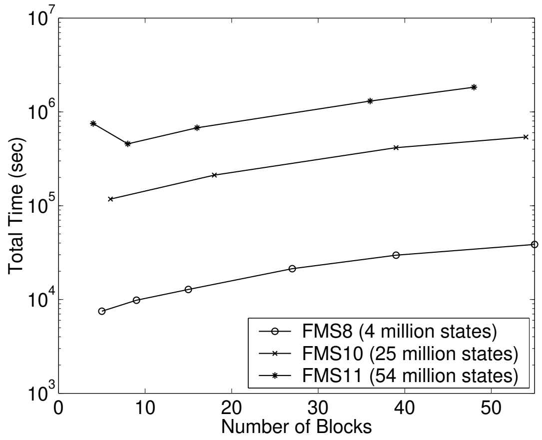
The run times of our symbolic out-of-core solution are recorded in column 8 under *outcore*. In order to measure the relative performance of our out-of-core algorithm, we ran PRISM on the same models and machines using the in-core hybrid MTBDD engine and the resulting run times are listed in column 7 under *incore*; Figure 3(d) represents this relative performance. All the solution times reported are real times. The last column indicates the amount of memory required to store the probability vector (n doubles). The run time for FMS ($l = 12$) was taken after running 100 iterations; we were not able to wait for its convergence, and hence the total number of iterations is not reported in the table.

The symbolic out-of-core algorithm has also been tested on machine2, a single processor machine with a relatively slower CPU and smaller RAM than machine1. This is to demonstrate the effectiveness of the algorithm on average capacity workstations, and in particular on a single CPU machine. Table 3 lists the performance measures of the algorithm on machine2, which are identical to the measures reported in Table 2. We observe that the ratio between the *outcore* run times listed in Table 2 and Table 3 reflects the ratio (1600MHz/440MHz) between the speeds of the two processors.

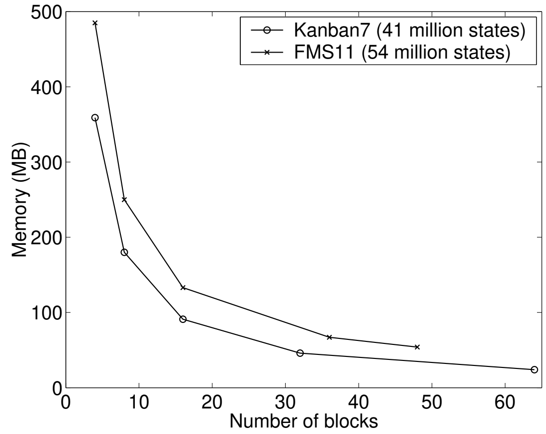
The symbolic out-of-core algorithm has been extensively tested and analysed; Figure 3 displays various performance characteristics of the algorithm. Figures 3(a), 3(b) and 3(e) depict the effect of increasing the number of blocks that the probability vector is divided into on the total solution times for Kanban and FMS system respectively. This solution time effect versus the number of blocks has been plotted for various sizes of CTMC matrices and a consistent pattern has been observed. For example, in Figure 3(b), the proportional increase in solution time for FMS ($l = 11$) is 2.4, although the proportional increase in the number of vector blocks is 12. We also observe that the pro-



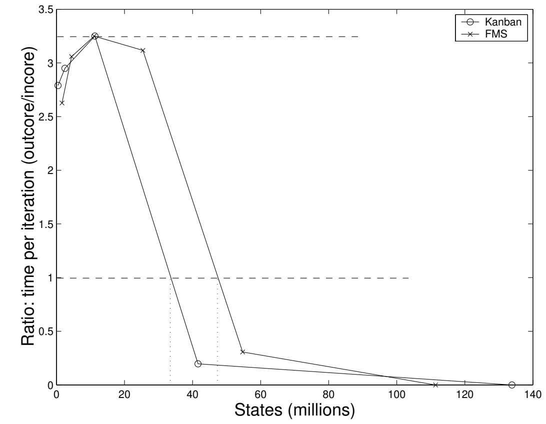
(a) Total times vs. blocks for Kanban



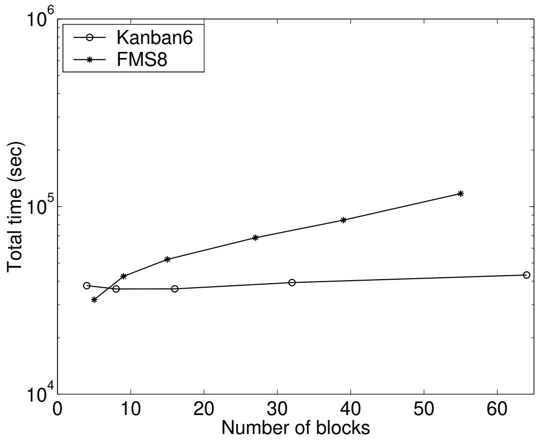
(b) Total times vs. blocks for FMS



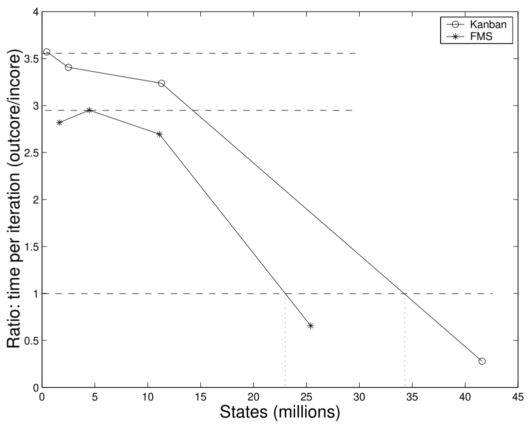
(c) Memory usage for Kanban and FMS



(d) In-core vs. out-of-core on machine1



(e) Total times vs. blocks on machine2



(f) In-core vs. out-of-core on machine2

Fig. 3. Performance graphs of the symbolic out-of-core solution

portional increase in total time decreases as the size of the matrix increases (for example, 5.1 for FMS ($l = 8$) reduces to 2.4 for FMS ($l = 11$)). Figure 3(c) displays the relationship between the memory requirement of the solution for various numbers of blocks. As expected, the memory requirement of the out-of-core solution decreases with an increase in the number of blocks.

In Figure 3(d) we have plotted the ratio of the in-core and out-of-core run times listed in Table 2 (on machine1), against the number of states for both models. The dashed horizontal line in this plot indicates the ideal behaviour ($incore = outcore$) desirable for out-of-core solution, and the vertical dotted line indicates the number of states where the out-of-core solution supposedly (because of thrashing for in-core solution) reaches the ideal behaviour. We note that the maximum slow-down is 3.24, and that the performance of the algorithm improves with the increase in the number of states. Figure 3(f) plots the same information but this time for machine2 (Table 3).

6 Conclusion

A new symbolic out-of-core algorithm has been introduced in this paper along with its implementation and a detailed analysis. The effectiveness of the algorithm has been demonstrated by testing it on two large models, and by solving models with up to 133 million states. It is evident that even larger systems can be solved by dividing the probability vector into a larger number of blocks. The paper has extended the limits on the size of models that are solvable on a single workstation by relaxing the limitations of the hybrid MTBDD approach of [19], which was caused by the need to store probability vectors explicitly. This approach is equally applicable to other symbolic methods such as Matrix Diagrams as well as the Kronecker methods, which have also been hindered by the same limitations.

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.

References

[1] Bahar, I., E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo and F. Somenzi, *Algebraic decision diagrams and their applications*, in: *Proc. ICCAD'93*, Santa Clara, 1993, pp. 188–191, also available in *Formal Methods in System Design*, 10(2/3), 1997.

[2] Bell, A. 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, 2001.

[3] Buchholz, P. and P. Kemper, *Compact representations of probability distributions in the analysis of superposed GSPNs*, in: R. German and B. Haverkort, editors, *Proc. Petri Nets and Performance Models (PNPM'01)*, Aachen, Germany, 2001, pp. 81–90.

[4] Ciardo, G., *What a Structural World*, in: R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, 2001, pp. 3–16.

[5] Ciardo, G. and A. Miner, *A Data Structure for the Efficient Kronecker Solution of GSPNs*, in: *Proc. 8th International Workshop on Petri Nets and Performance Models (PNPM'99)*, Zaragoza, 1999.

[6] Ciardo, G. and M. Tilgner, *On the use of Kronecker Operators for the Solution of Generalized Stocastic Petri Nets*, ICASE Report 96-35, Institute for Computer Applications in Science and Engineering (1996).

[7] Ciardo, G. and K. S. Trivedi, *A Decomposition Approach for Stochastic Reward Net Models*, Performance Evaluation **18** (1993), pp. 37–59.

[8] Clarke, E., 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)*, Tahoe City, 1993, pp. 6a:1–15, also available in *Formal Methods in System Design*, 10(2/3), 1997.

[9] Deavours, D. D. and W. H. Sanders, *An Efficient Disk-based Tool for Solving Very Large Markov Models*, in: *Lecture Notes in computer Science: Proceedings of the 9th International Conference on Modelling Techniques and Tools (TOOLS '97)*, Springer-Verlag, St. Malo, France, 1997, pp. 58–71.

[10] Deavours, D. D. and W. H. Sanders, *An Efficient Disk-based Tool for Solving Large Markov Models*, Performance Evaluation **33** (1998), pp. 67–84.

[11] Deavours, D. D. and W. H. Sanders, *"On-the-fly" Solution Techniques for Stochastic Petri Nets and Extensions*, IEEE Transactions on Software Engineering **24** (1998), pp. 889–902.

[12] Duff, I. S., A. M. Erisman and J. K. Reid, "Direct Methods for Sparse Matrices," Oxford Science Publications, Clarendon Press Oxford, (with corrections)1997.

[13] Hachtel, G., E. Macii, A. Pardo and F. Somenzi, *Markovian analysis of large finite state machines*, IEEE Transactions on CAD **15** (1996), pp. 1479–1493.

[14] Hermanns, H., M. Kwiatkowska, G. Norman, D. Parker and M. Siegle, *On the use of MTBDDs for performability analysis and verification of stochastic systems*, Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems (2002), to appear.

[15] Hermanns, H., 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.

[16] Knottenbelt, W. J. and P. G. Harrison, *Distributed Disk-based Solution Techniques for Large Markov Models*, in: *Proc. Numerical Solution of Markov Chains (NSMC'99)*, Zaragoza, 1999.

[17] Kwiatkowska, M. and R. Mehmood, *Out-of-Core Solution of Large Linear Systems of Equations arising from Stochastic Modelling*, in: *Proc. Process Algebra and Probabilistic Methods (PAPM-PROBMIV'02)*, July 2002, available as Volume 2399 of *LNCS*.

[18] Kwiatkowska, M., G. Norman and D. Parker, *PRISM: Probabilistic Symbolic Model Checker*, in: *Proc. TOOLS 2002*, April 2002, available as Volume 2324 of *LNCS*.

[19] Kwiatkowska, M., 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*.

[20] Miner, A. S., *Efficient Solution of GSPNs using Canonical Matrix Diagrams*, in: R. German and B. Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, 2001, pp. 101–110.

[21] Parker, D., "Implementation of symbolic model checking for probabilistic systems," Ph.D. thesis, University of Birmingham (2002), to appear.

[22] Plateau, B., *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*, Austin, TX, USA, 1985, pp. 147–153.

[23] *PRISM web page*, `http://www.cs.bham.ac.uk/~dxp/prism/`.

[24] Saad, Y., "Iterative Methods for Sparse Linear Systems," PWS Publishing Company, 1996.

[25] Siegle, M., *Advances in Model Representations*, in: L. de Alfaro and S. Gilmore, editors, *Proc. PAPM/PROBMIV 2001, Available as Volume 2165 of* LNCS (2001), pp. 1–22.

[26] Stewart, W. J., "Introduction to the Numerical Solution of Markov Chains," Princeton University Press, 1994.

[27] Toledo, S., *A Survey of Out-of-Core Algorithms in Numerical Linear Algebra*, in: J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society Press, Providence, RI, 1999 .