Serial Disk-based Analysis of Large Stochastic Models

Rashid Mehmood

School of Computer Science, University of Birmingham,
Birmingham B15 2TT, United Kingdom
R.Mehmood@cs.bham.ac.uk

Abstract. The paper presents a survey of out-of-core methods available for the analysis of large Markov chains on single workstations. First, we discuss the main sparse matrix storage schemes and review iterative methods for the solution of systems of linear equations typically used in disk-based methods. Next, various out-of-core approaches for the steady state solution of CTMCs are described. In this context, serial out-of-core algorithms are outlined and analysed with the help of their implementations. A comparison of time and memory requirements for typical benchmark models is given.

1 Introduction

Computer and communication systems are being used in a variety of applications ranging from stock markets, information services to aeroplane and car control systems. Discrete-state models have proved to be a valuable tool in the analysis of these computer systems and communication networks. Modelling of such systems involves the description of the system's behaviour by the set of different states the system may occupy, and identifying the transition relation among the various states of the system. Uncertainty is an inherent feature of real-life systems and, to take account of such behaviour, probability distributions are associated with the possible events (transitions) in each state, so that the model implicitly defines a stochastic process. If the probability distributions are restricted to be either geometric or exponential, the stochastic process can be modelled as a discrete time (DTMC) or a continuous time (CTMC) Markov chain respectively. A Markov decision process (MDP) admits a number of discrete probability distributions enabled in a state which are chosen nondeterministically by the environment. We concentrate in this paper on continuous time Markov chains.

A CTMC may be represented by a set of states and a transition rate matrix Q containing state transition rates as coefficients. The matrix coefficients determine transition probabilities and state sojourn times. A CTMC is usually stored as a sparse matrix, where only the nonzero entries are stored. In general, when analysing CTMCs, the performance measure of interest corresponds to either the probability of being in a certain state at a certain time (*transient*) or the long-run (*steady state*) probability of being in a state. Transient state probabilities can be determined by solving a system of ordinary differential equations. The computation of steady state probabilities involves the solution of a system of linear equations. We focus in this paper on the steady state solution of a CTMC.

The overall process of the state based analytical modelling for CTMCs involves the *specification* of the system, the *generation* of the state space, and the *numerical computation* of all performance measures of interest. Specification of a system at the level of a Markov chain, however, is difficult and error-prone. Consequently, a wide range of high-level formalisms have been developed to specify abstractions of such systems. These formalisms, among others, include queueing networks (QN) [CG89], stochastic Petri nets (SPN) [Mol82], generalised SPNs (GSPN) [MBC84,MBC+95], stochastic process algebras (SPA), such as PEPA [Hil94], EMPA [BG96] and TIPP [HR94], mixed forms such as queueing Petri nets (QPN) [Bau93], and stochastic automata networks (SAN) [Pla85,PA91]. See for example [Sie01], for a survey of model representations.

Once a system is specified using some suitable high-level formalism, the entire[1] state space needs to be generated from this specification. In this paper, we do not discuss state space generation algorithms and techniques.

The size of a system (number of states) is typically exponential in the number of parallel components in the system. This problem is known as the *state-space explosion* or the *largeness* problem, and has motivated researchers to tread a number of directions for the numerical solution of a Markov chain. These can be broadly classified into *explicit* and *implicit* techniques. The so-called explicit methods store a CTMC explicitly, using a data structure of size proportional to the number of states and transitions. The implicit methods, on the other hand, use some kind of symbolic data structure for the storage of a CTMC.

The focus of this paper is the serial disk-based performance analysis of CTMCs, and hence we will discuss here those *out-of-core*[2] techniques that store all or a part of the data structure on disk for the numerical solution of a CTMC. The term *in-core* is used when the data is in the main memory of a computer. The term "serial" indicates that the numerical computations are performed on a single processor, as opposed to in "parallel", where the computational task is distributed between a number of processors.

The paper is organised as follows. The numerical solution methods used in the serial disk-based approaches are discussed in Section 2. A compact sparse matrix representation is at the heart of the analysis techniques for large Markov chains, especially considering that the time required for disk read/write determines the overall solution time for out-of-core methods. The main storage schemes for sparse matrices are reviewed in Section 3. The out-of-core algorithms are presented in Section 4. These algorithms are further analysed in Section 5 with the help of results of their implementations applied to typical benchmark models. We conclude with Section 6.

[1] With the exception of product-form queueing networks [BCMP75]. *On-the-fly* techniques [DS98b], to some extent, are also an exception.

[2] Algorithms that are designed to achieve high performance when their data structures are stored on disk are known as *out-of-core algorithms*; see [Tol99], for example.

2 Numerical Methods

Performance measures for stochastic models are traditionally derived by generating and solving a Markov chain obtained from some high-level formalism. Let $Q \in \mathbb{R}^{n \times n}$ be the infinitesimal generator matrix; the order of Q equals the number of states in the CTMC. The off-diagonal elements of the matrix 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 element q_{ij} gives the rate of moving from state i to state j. The matrix Q is usually sparse; further details about the properties of these matrices can be found in [Ste94]. The steady state behaviour of a CTMC is given by:

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

where π is the steady state probability vector. A sufficient condition for the unique solution of the equation (1) is that the CTMC is finite and irreducible. A CTMC is *irreducible* if every state can be reached from every other state [Ste94]. For the remainder of the paper, we restrict attention to solving only irreducible CTMCs; for details of the solution in the general case see [Ste94]. The equation (1) can be reformulated as $Q^T \pi^T = 0$, and hence 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 matrix entries during the factorisation phase. Iterative methods do not modify matrix A; rather, they involve the matrix only in the context of the matrix-vector product (MVP). The term "iterative methods" refers to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step [BBC+94].

Before we move on to the next section, where we discuss the basic iterative methods for the solution of the system of equations $Ax = b$, we mention the *Power method*. Given the generator matrix Q, setting $P = I + Q/\alpha$ in Equation (1), where $\alpha \geq max_i \mid q_{ii} \mid$, leads to:

$$\pi P = \pi. \tag{2}$$

Using $\pi^{(0)}$ as the initial estimate, an approximation of the steady state probability vector after k transitions is given by $\pi^{(k)} = \pi^{(k-1)} P$. This method successively multiplies the steady state probability vector with the matrix P until convergence is reached. It is guaranteed to converge (for irreducible CTMCs), though convergence can be very slow. Below we consider alternative iterative methods, which may fail to converge for some models but, in practice, converge much faster than the Power method.

2.1 Basic Iterative Methods

We discuss, in this section, the so-called *stationary iterative methods* for the solution of the system of equations $Ax = b$, that is, the methods that can be expressed in the simple form $x^{(k)} = Bx^{(k-1)} + c$, where neither B nor c depend upon the iteration count k [BBC+94]. Beginning with a given approximate solution, these methods modify the components of the approximation in each iteration, until a required accuracy is achieved. In the k-th iteration of the *Jacobi method*, for example, we calculate element-wise:

$$x_i^{(k)} = a_{ii}^{-1}\left(b_i - \sum_{j\neq i} x_j^{(k-1)} a_{ij}\right), \quad 0 \leq i < n, \tag{3}$$

where a_{ij} denotes the element in row i and column j of matrix A. It can be seen in equation (3) that the new approximation of the iteration vector ($x^{(k)}$) is calculated using only the old approximation of the vector ($x^{(k-1)}$). This makes the Jacobi method suitable for parallelisation. However, Jacobi method exhibits slow convergence.

The *Gauss-Seidel* (GS) iterative method, which in practice converges faster than the Jacobi method, uses the most recently computed approximation of the solution:

$$x_i^{(k)} = a_{ii}^{-1}\left(b_i - \sum_{j<i} x_j^{(k)} a_{ij} - \sum_{j>i} x_j^{(k-1)} a_{ij}\right), \quad 0 \leq i < n. \tag{4}$$

Another advantage of the Gauss-Seidel method is that it can be implemented with a single iteration vector, whereas the Jacobi method requires two.

Finally, we mention the *successive over-relaxation* (SOR) method. An SOR iteration is of type

$$x_i^{(k)} = \omega \hat{x}_i^{(k)} + (1-\omega) x_i^{(k-1)}, \quad 0 \leq i < n, \tag{5}$$

where $\hat{x}$ denotes a Gauss-Seidel iterate, and $\omega \in (0,2)$ is a relaxation factor. The method is under-relaxed for $0 < \omega < 1$, and is over-relaxed for $\omega > 1$; the choice $\omega = 1$ reduces SOR to Gauss-Seidel. It is shown in [Kah58] that SOR fails to converge if $\omega \notin (0,2)$. For a good choice of ω, SOR can have considerably better convergence behaviour than Gauss-Seidel, but unfortunately a priori computation of an optimal value for ω is not feasible.

2.2 Block Iterative Methods

Consider a partitioning of the state space S of a CTMC into B contiguous partitions $S_0, \ldots, S_{B-1}$ of sizes $n_0, \ldots, n_{B-1}$, such that $n = \sum_{i=0}^{B-1} n_i$. Using this, the matrix A can be divided into B^2 blocks, $\{A_{ij} \mid 0 \leq i, j < B\}$, where the rows and columns of block A_{ij} correspond to the states in S_i and S_j, respectively, i.e., block A_{ij} is of size $n_i \times n_j$. We also define $n_{\max} = \max\{n_i \mid 0 \leq i < B\}$. Using such a partitioning of the state space for $B = 4$, the system of equations

$Ax = b$ can be partitioned as:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} \tag{6}$$

Given the partitioning introduced above, block iterative methods essentially involve the solution of B sub-systems of linear equations of sizes $n_0, \ldots, n_{B-1}$ within a *global* iterative structure, say Gauss-Seidel. Hence, from (4) and (6), the *block Gauss-Seidel* method for the solution of the system $Ax = b$ is given by:

$$X_i^{(k)} = A_{ii}^{-1} \left(B_i - \sum_{j<i} A_{ij}\, X_j^{(k)} - \sum_{j>i} A_{ij}\, X_j^{(k-1)} \right), \quad 0 \le i < B, \tag{7}$$

where $X_i^{(k)}$, $X_i^{(k-1)}$ and B_i are the i-th blocks of vectors $x^{(k)}$, $x^{(k-1)}$ and b respectively, and, as above, A_{ij} denotes the (i, j)-th block of matrix A. Hence, in each of the B phases of the k-th iteration of the block Gauss-Seidel iterative method, we solve the equation (7) for $X_i^{(k)}$. These sub-systems can be solved by either direct or iterative methods. It is not necessary even to use the same method for each sub-system. If iterative methods are used to solve these sub-systems then we have several *inner* iterative methods within a global or *outer* iterative method. Each sub-system of equations can receive either a fixed or varying number of *inner* iterations. Such block methods (with an inner iterative method) typically require fewer iterations, but each iteration requires more work (multiple inner iterations). Block iterative methods are well known and are an active area of research; see [Ste94], for example.

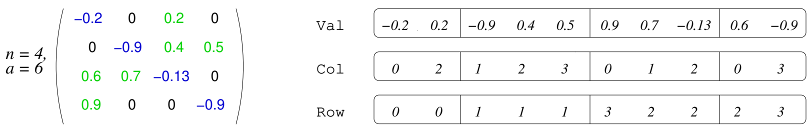
2.3 Test for Convergence

Usually, a test for convergence is carried out in each iteration of an iterative method and the method is stopped when the convergence criterion is met; [BBC$^+$94] discusses this subject in some details. In the context of the steady state solution of a CTMC, a widely used test is the relative error criterion:

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

3 Matrix Storage Considerations

An $n \times n$ dense matrix is usually stored in a two-dimensional $n \times n$ array. For sparse matrices, in which most of the entries are zero, storage schemes are sought which can minimise the storage while keeping the computational costs to a minimum. Consequently, a number of sparse storage schemes exist which exploit various matrix properties, e.g., the *sparsity pattern* of a matrix. The sparse



(a) A CTMC matrix (b) The coordinate format

Fig. 1. A 4×4 sparse matrix and its storage in the coordinate format

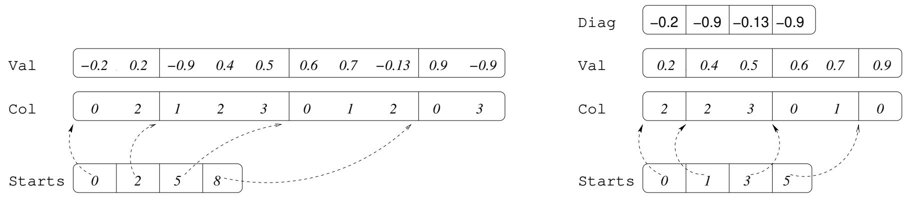
schemes we will discuss in this section are not exhaustive; for more schemes see, for instance, [BBC+94].

The simplest of sparse schemes which makes no assumption about the matrix is the so-called *coordinate format* [Saa90,Her92]. Figure 1 gives an 4×4 sparse matrix with a off-diagonal nonzero entries and its storage in coordinate format. The scheme uses three arrays. The first array `Val` (of size $a + n$ doubles) stores the matrix nonzero entries in any order, while the arrays `Col` and `Row`, both of size $a + n$ ints, store the column and row indices for these entries, respectively. Given an 8-byte floating point number representation (double) and a 4-byte integer representation (int), the coordinate format requires $16(a + n)$ bytes to store the whole sparse matrix, including diagonal and off-diagonal entries.

Figure 2(a) illustrates the storage of the matrix in Figure 1(a) in the *compressed sparse row* (CSR) [Saa90] format. All the $a+n$ nonzero entries are stored row by row in the array `Val`, while `Col` contains column indices of these nonzero entries; the elements within a row can be stored in any order. The i-th element of the array `Starts` (of size n ints) contains the index in `Val` (and `Col`) of the beginning of the i-th row. The CSR format requires $12a + 16n$ bytes to store the whole sparse matrix including the diagonal.

3.1 Separate Diagonal Storage

A second dimension for sparse schemes is added if we consider that many iterative algorithms treat diagonal entries of a matrix differently. This gives us the following two additional choices for sparse storage schemes; an alternative choice will be mentioned in Section 3.2.



(a) The CSR format (b) The MSR format

Fig. 2. Storage in the CSR and MSR formats of the matrix in Figure 1(a)

Case A: The diagonal entries may be stored separately in an array of n doubles. Storage of column indices of diagonal entries in this case is not required, which gives us a saving of $4n$ bytes over the CSR format. This scheme is known as the *modified sparse row* (MSR) format [Saa90] (a modification of CSR), see Figure 2(b). We note that the MSR scheme essentially is the same as the CSR format except that the diagonal elements are stored separately. The scheme requires $12(a + n)$ bytes to store the whole sparse matrix. Some computational advantage may be obtained by storing the diagonal entries as $1/a_{ii}$ instead of a_{ii}, which replaces n division operations with n multiplications.

Case B: In certain contexts, such as for the steady state solution of a Markov chain, it is possible to avoid the *in-core* storage of the diagonal entries. Given $R = Q^T D^{-1}$, the system $Q^T \pi^T = 0$ can be equivalently written as $Q^T D^{-1} D \pi^T = Ry = 0$, with $y = D\pi^T$. The matrix D is defined as the diagonal matrix with $d_{ii} = q_{ii}$, for $0 \le i < n$. Consequently, the equivalent system $Ry = 0$ can be solved with all the diagonal entries of the matrix R being 1. The original diagonal entries can be stored on disk for computing π from y. This saves $8n$ bytes of the in-core storage, along with computational savings of n divisions per each step in an iterative method such as Gauss-Seidel.

3.2 Exploiting Matrix Properties

The number of distinct values in a generator matrix depends on the model. This characteristic can lead to significant memory savings if one considers indexing the nonzero entries in the above mentioned formats. Consider the MSR format. Let $MaxD$ be the number of distinct values the off-diagonal entries of a matrix can take, where $MaxD \le 2^{16}$; then $MaxD$ distinct values can be stored as `double Val`$[MaxD]$. The indices to this array of distinct values cannot exceed 2^{16}, and, in this case, the array `double Val`$[a]$ in MSR format can be replaced with `short Val_i`$[a]$. In the context of CTMCs, in general, the maximum number of entries per row of a generator matrix is also small, and is limited by the maximum number of transitions leaving a state. If this number does not exceed 2^8, the array `int Starts`$[n]$ in MSR format can be replaced by the array `char row_entries`$[n]$.

Consequently, in addition to the array of distinct values, `Val`$[MaxD]$, the indexed variation of MSR mentioned above uses three arrays: the array `Val_i`$[a]$ of length $2a$ bytes for the storage of a short (2-byte integer representation) indices to the $MaxD$ entries, an array of length $4a$ bytes to store a column indices as int (as in MSR), and the n-byte long array `row_entries`$[n]$ to store the number of entries in each row. The total in-core memory requirement for this scheme is $6a + n$ bytes plus the storage for the actual distinct values in the matrix. Since the storage for the actual distinct values is relatively small for large models, we do not consider it in future discussions. Such variations of the MSR format, based on indexing the matrix elements, have been used in the literature [DS98a,Bel99,KH99,BH01,KM02] under different names. We call it the *indexed MSR* format.

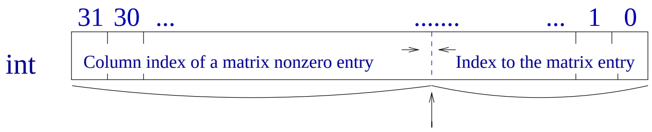


Fig. 3. Storage of an index to a matrix distinct value using available space in int

We note that, in general, for any of the above-mentioned formats, it is possible to replace the array `double Val`$[a]$ with `short Val_i`$[a]$, or with `char Val_i`$[a]$, if $MaxD$ equals 2^{16}, or 2^8, respectively. In fact, $\lceil log_2(MaxD) \rceil$ bits suffice for each index. Similarly, it is also possible to index diagonal entries of a matrix provided the diagonal vector has relatively few distinct entries. This justifies an alternative choice for separate diagonal storage (see Section 3.1).

We describe here another variation of the MSR scheme. It has been mentioned that the indexed MSR format exploits the fact that the number of entries in a generator matrix is relatively small. We have also seen that the indexed MSR format (this is largely true for all the formats considered here) stores the column index of a nonzero entry in a matrix as an int. An int usually uses 32 bits, which can store a column index as large as 2^{32}. The size of the models which can be stored within the RAM of a modern workstation are much smaller than 2^{32}. For example, it is evident from Table 1 that using an in-core method such as Gauss-Seidel, 54 million states is a limit for solving a CTMC on a typical single workstation. The column index for a model with 54 million states requires at most 26 bits, leaving 6 bits unused. Even more bits can be made available if we consider that, for out-of-core and for parallel solutions, it is common practice (or, at least it is possible) to use local numbering for a column/row index inside each matrix block.

The *compact MSR* format [KM02] exploits the above mentioned facts and stores the column index of a matrix entry along with the index to the actual value of this entry in a single int. This is depicted in Figure 3. The storage and retrieval of these indices into, and from, an int can be carried out efficiently using bit operations. Since the operation $Q^T D^{-1}$ increases the number of distinct values in the resulting matrix R, matrix Q and the diagonal vector d can be stored separately. As mentioned earlier, the diagonal entries can be indexed, and the distinct entries can be stored as $1/a_{ii}$ to save n divisions per iteration; indices to these distinct values may be stored as short. The compact MSR scheme uses three arrays: the array `Col_i`$[a]$ of length $4a$ bytes which stores the column positions of matrix entries as well as the indices to these entries, the n-byte long array `row_entries`$[n]$ to store the number of entries in each row, and the $2n$-byte long array `Diag_i`$[n]$ of short indices to the original values in the diagonal. We do not consider the storage for the original matrix entries. The total memory requirements in the compact MSR format is thus $4a+3n$ bytes, around 30% more compact than the indexed MSR format.

Table 1 summarises a fact sheet for the model of a flexible manufacturing system (FMS) [CT93] comparing storage requirements in MSR, indexed MSR, and compact MSR formats. The first column in Table 1 gives the parameter k

k	states (n)	off-diagonal nonzero (a)	a/n	memory required for Q (MB)			MB
				MSR format	*Indexed MSR*	*Compact MSR*	per π
8	4,459,455	38,533,968	8.64	489	225	160	34
9	11,058,190	99,075,405	8.96	1,260	577	409	84
10	25,397,658	234,523,289	9.23	2,962	1,366	967	194
11	54,682,992	518,030,370	9.47	6,554	3,016	2,133	417
12	111,414,940	1,078,917,632	9.68	13,563	6,279	4,433	850
13	216,427,680	2,136,215,172	9.87	32,361	15,149	10,580	1,651

Table 1. Storage requirements for the FMS models in various formats

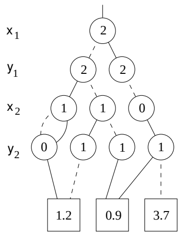
(number of tokens in the system); the second and third columns list the resulting number of reachable states and the number of transitions respectively. The number of states and the number of transitions increase with an increase in the parameter k. The largest model generated is FMS ($k = 13$) with over 216 million reachable states and 2.1 billion transitions. The fourth column (a/n) gives the average number of the off-diagonal nonzero entries per row, which serves as an indication of the matrix sparsity. Columns 5–7 give storage requirements for the matrices in MB for the MSR, the indexed MSR and the compact MSR schemes respectively. Finally, the last column lists the memory required to store a single iteration vector of doubles (8 bytes) for the solution phase.

3.3 Alternatives for Matrix Storage

We have discussed various explicit sparse matrix schemes for CTMCs. Another approach, which has been very successful in model checking, is the implicit storage of the matrix. The term implicit methods is used because these data structures do not require size proportional to the number of states. Implicit methods include multi-terminal binary decision diagrams (MTBDDs) [CFM$^+$93,BFG$^+$93], the Kronecker approach [Pla85], matrix diagrams (MDs) [CM99] and on-the-fly methods [DS98b]. A brief explanation of MTBDDs, for which a vector out-of-core implementation will be discussed in 4.5, is given in the following paragraph; further discussion of the implicit methods is beyond the scope of this paper, and the interested reader is invited to consult the individual references, or see [MP03,BK03] for recent surveys of such data structures.

Multi-Terminal Binary Decision Diagrams (MTBDDs)
MTBDDs [CFM$^+$93,BFG$^+$93] are an extension of binary decision diagrams (BDDs). An MTBDD is a rooted, directed acyclic graph, which represents a function mapping Boolean variables to real numbers. MTBDDs can be used to encode real-valued matrices (and vectors) by encoding their indices as Boolean variables. The prime reason for using MTBDDs is that they can provide extremely compact storage for the generator matrices of very large CTMCs, provided that structure and regularity derived from their high-level description can be exploited. Here, we describe a slight variant of MTBDDs called *offset-labelled MTBDDs* [KNP02b,Par02]. This data structure additionally allows information about which states of a CTMC are reachable. Techniques which use data structures based on BDDs are often called *symbolic* approaches.

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



Entry of the matrix	Encoding x1 x2	y1 y2	MTBDD Path x1 y1 x2 y2
$(0,1) = 0.9$	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. An offset-labelled MTBDD representation of a matrix

Figure 4 shows a matrix, its representation as an offset-labelled MTBDD, and a table explaining how the information is encoded. To preserve structure in the symbolic representation of a CTMC's generator matrix, its diagonal elements are stored separately as an array. Hence, the diagonal of the matrix in Figure 4 is shown to be zero. The offset-labelled MTBDD in the figure represents a function over four Boolean variables x_1, y_1, x_2, x_2. For a given valuation of these variables, the value of this function can be computed by tracing a path from the top of the offset-labelled MTBDD to the bottom, taking the dotted edge if the Boolean variable on that level is 0 and the solid edge if the variable is 1. The value can be read from the label of the bottom (terminal) node reached. For example, if $x_1 = 1, y_1 = 0, x_2 = 1, y_2 = 1$, the function returns 0.9.

To represent the matrix, the offset-labelled MTBDD in Figure 4 uses the variables x_1, x_2 to encode row indices and y_1, y_2 to encode column indices. Notice that these are ordered in an interleaved fashion in the figure. This is a common heuristic in BDD-based representations to reduce their size. In the example, row and column indices are encoded using the standard binary representation of integers. For example, the row index 3 is encoded as 11 ($x_1 = 1, x_2 = 1$) and the column index 1 is encoded as 01 ($y_1 = 0, y_2 = 1$). To determine the value of the matrix entry, we read the value of the function represented by the MTBDD for $x_1 = 1, y_1 = 0, x_2 = 1, y_2 = 1$. Hence, the matrix entry $(3, 1)$ is 0.9.

The integer values on the nodes of the data structure are *offsets*, used to compute the *actual* row and column indices of the 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*. The actual row index is determined by summing the offsets on x_i nodes from which the solid edge is taken (i.e. if $x_i = 1$). The actual column index is computed similarly for y_i nodes. In the example in Figure 4, state 2 is not reachable. For the previous example of matrix entry (3,1), the actual row index is 2+0=2 and the column index is 1 (using only the offset on the y_2 level).

Note that each node of an MTBDD can be seen to represent a submatrix of the matrix represented by the whole MTBDD. Since an MTBDD is based on binary decisions, descending each level (each pair of x_i and y_i variables) of the data structure splits the matrix into 4 submatrices. Hence, descending l levels, gives a decomposition into $(2^l)^2$ blocks. For example, descending one level of the MTBDD in Figure 4, gives a decomposition of the matrix into 4 blocks and we see that each x_2 node represents a 2×2 submatrix of the 4×4 matrix. This allows convenient and fast access to individual submatrices. However, since the

MTBDD actually encodes a matrix over its potential state space and the distribution of the reachable states across the state space is unpredictable, descending l levels of the MTBDD actually results in blocks of varying and uneven sizes.

Numerical solution of CTMCs can be performed purely using MTBDDs (see e.g. [HMKS99]). This is done by representing both the matrix and the vector as MTBDDs and using an MTBDD-based matrix-vector multiplication algorithm ([CFM$^+$93,BFG$^+$93]). However, this approach is often very inefficient because the MTBDD representation of the vector is irregular and grows quickly during solution. A better approach in general is to use offset-labelled MTBDDs to store the matrix and an array to store the vector ([KNP02b,Par02]). Computing the *actual* row and column indices of matrix entries is important in this case because they are needed to access the elements of the array storing the vector. All matrix entries can be extracted from an offset-labelled MTBDD in a single pass using a recursive traversal of the data structure.

4 Out-of-Core Iterative Solution Methods

We survey here the serial out-of-core methods for the steady state solution of CTMCs, i.e., the methods which store whole or part of the data structure on disk. We begin with an in-core block Gauss-Seidel algorithm in the next section. We then present and explain a matrix and a complete out-of-core Gauss-Seidel algorithm in Sections 4.2 and 4.4, respectively. The matrix out-of-core approach of [DS98a] is described in Section 4.3, and the symbolic out-of-core solution of [KMNP02] is discussed in Section 4.5.

4.1 The In-Core Approach

We give an in-core block Gauss-Seidel algorithm for the solution of the system $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. Block iterative methods are described in Section 2.2 and the block Gauss-Seidel method was formulated in equation (7).

A typical iteration of the block Gauss-Seidel method is shown in Figure 5. The algorithm requires an array x of size n (number of states) to store the iteration vector, the i-th block of which is denoted X_i, and another array $\tilde{X}_i$ of size $n_{\max}$ to accumulate the sub-MVPs, $A_{ij}X_j$, i.e., the multiplication of a single matrix block by a single vector block; see Section 2.2. The subscript i of $\tilde{X}_i$ 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 B such arrays.

Each iteration of the algorithm is divided into B phases. In the i-th phase, the method updates elements in the i-th block of the iteration vector. The update of the i-th block, X_i, only requires access to entries from the i-th row of blocks in A, i.e., A_{ij} for $0 \leq j < B$. This is illustrated in Figure 6 for $B = 4$ and $i = 1$; the matrix and vector blocks used in the calculation are shaded grey. In the figure, all blocks are of equal size but this is generally not the case. Line 5 of the algorithm in Figure 5 performs a unit of computation, a sub-MVP, and accumulates these products. Finally, line 6 corresponds to solving a system of

for $i = 0$ to $B - 1$
    $\tilde{X}_i \leftarrow 0$
    for $j = 0$ to $B - 1$
        if $j \neq i$
            $\tilde{X}_i = \tilde{X}_i - A_{ij} X_j$
    Solve $A_{ii} X_i = \tilde{X}_i$
    Test for convergence
Stop if converged

Fig. 5. An iteration of the block Gauss-Seidel algorithm

equations, either by direct or iterative methods (see Section 2.2). We use the Gauss-Seidel iterative method to solve $A_{ii} X_i = \tilde{X}_i$. More precisely, we apply the following to update each of the n_i elements of the i-th vector block, X_i:

for $p = 0$ to $n_i - 1$
    $A_{ii}[p,p]\ X_i[p] = \tilde{X}_i[p] \ - \ \sum_{q \neq p} \ A_{ii}[p,q]\ X_i[q],$

where $X_i[p]$ is the p-th entry of the vector block X_i, and $A_{ii}[p,q]$ denotes the (p,q)-th element of the diagonal block (A_{ii}). We note that applying one *inner* Gauss-Seidel iteration for each X_i in the global Gauss-Seidel iterative structure reduces the block Gauss-Seidel method to the standard Gauss-Seidel method, although the method is based on block sub-MVPs.

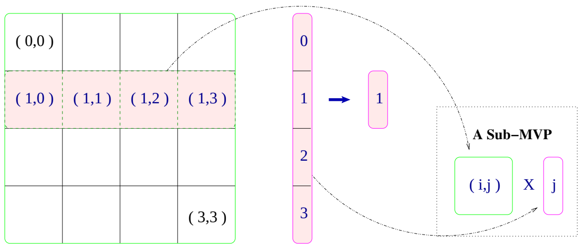


Fig. 6. Matrix vector multiplication at block level

4.2 A Matrix Out-of-Core Approach

In an implementation of the block Gauss-Seidel algorithm mentioned in the previous subsection, the matrix can be stored using some sparse storage scheme (see Section 3) and the vector 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 is to store the matrix on disk and read blocks of matrix into RAM when required. In each iteration of an iterative method, we can do the following:

while there is a block to read
    read a matrix block
    do computations using the matrix and vector blocks

Integer constant: B (*number of vector blocks*)
Semaphores: S_1, S_2: occupied
Shared variables: R_0, R_1 (*To read matrix A blocks into RAM*)

DiskIO Process

1. Local variables: i, j, $t = 0$
2. **while** not converged
3.   **for** $i \leftarrow 0$ to $B-1$
4. $R_t = \texttt{read}(\ A_{ij},\ j = 0 : B-1\ )$
5.   Signal(S_1)
6.   Wait(S_2)
7. $t = (t+1)\ mod\ 2$

Compute Process

1. Local variables: p, q, $t = 0$
2. **while** not converged
3.   **for** $i \leftarrow 0$ to $B-1$
4.   Wait(S_1)
5.   Signal(S_2)
6. $\tilde{X}_i = \texttt{subMVPs}(-\sum_{j \neq i}^{0:B-1} A_{ij} X_j,\ R_t)$
7. $\texttt{Solve}(\ A_{ii} X_i = \tilde{X}_i,\ R_t)$
8.   Test for convergence
9. $t = (t+1)\ mod\ 2$

Fig. 7. A matrix out-of-core block Gauss-Seidel algorithm

We note that, in this disk-based approach, the processor will remain idle until a block has been read into RAM. We also note that the next disk read operation will not be initiated until the computations have been performed. It is not an efficient approach, particularly for large models and iterative methods, because they require a relatively large amount of data per floating point operation. We would like to use a two-process approach, where the disk I/O and the computation can proceed concurrently. We begin with a basic out-of-core algorithm and explain its working. In the coming sections we discuss the out-of-core approaches pursued in the literature.

Figure 7 presents a high-level description of a typical, matrix out-of-core algorithm which uses the block Gauss-Seidel method for the solution of the system $Ax = 0$. The term "matrix out-of-core" implies that only the matrix is stored out-of-core and the vector is kept in-core. The algorithm is implemented using two separate concurrent processes: the *DiskIO Process* and the *Compute Process.* The two processes communicate via shared memory and synchronise with semaphores.

The algorithm of Figure 7 assumes that the (n-state) CTMC matrix to be solved is divided into B^2 blocks of size $n/B \times n/B$, and is stored on disk. For intuitive reasons, in Figure 7, the vector x is also shown to be divided into B blocks, although a single array of doubles is used to keep the whole vector. Another vector $\tilde{X}_i$ of size n/B is required to accumulate the sub-MVPs (line 6). The algorithm assumes that, before it commences, the array x holds an initial approximation to the solution.

Each iteration of the algorithm given in Figure 7 is divided into B phases, where the i-th phase computes the next approximation for the i-th block of the iteration vector. To update the i-th block, X_i, the *Compute Process* requires access to entries from the i-th row of blocks in A, i.e., A_{ij} for $0 \leq j < B$; see Figure 6. The *DiskIO Process* helps the *Compute Process* with this task and fetches from disk (line 4) the required row of the blocks in matrix A. The algorithm uses two shared memory buffers, R_0 and R_1, to achieve the communication between the two processes. At a certain point in time during the execution, the *DiskIO Process* is reading a block of matrix into one shared buffer, say R_0, while the *Compute Process* is consuming a matrix block from the other buffer, R_1. Both processes alternate the value of a local variable t between 0 and 1, in order to switch between the two buffers R_0 and R_1. The two semaphores S_1 and S_2 are used to synchronise the two processes, and to prevent inconsistencies.

The high-level structure of the algorithm, given in Figure 7, is that of a producer-consumer problem. In each execution of its `for` loop, the i-th phase (lines $3-7$), the *DiskIO Process* reads the i-th row of blocks in A, into one of the shared memory buffers R_t, and issues a Signal($\cdot$) operation on S_1 (line 5). Since the two semaphores are occupied initially, the *Compute Process* has to wait on S_1 (line 4). On receiving this signal, the *Compute Process* issues a return signal on S_2 (line 5) and then advances to update the i-th vector block (lines $6-7$). The *DiskIO process*, on receiving this signal from the *Compute Process*, advances to read the next i-th row of blocks in A. This activity of the two processes is repeated until all of the vector blocks have been updated; the processes then advance to the next iteration.

Implementation

We have used the compact MSR storage scheme (see Section 3) to store the matrix in our implementation of the matrix out-of-core algorithm. The blocks have been stored on disk in the order they are required during the numerical computation. Hence, the *DiskIO Process* is able to read the file containing the matrix sequentially throughout an iteration of the algorithm.

4.3 The Deavours and Sanders' Approach

Deavours and Sanders [DS97] were the first to consider a matrix out-of-core technique for the steady state solution of Markov models. They used the block Gauss-Seidel method in their tool to reduce the amount of disk I/O. They partitioned the matrix into a number of sub-systems (or blocks) and applied multiple Gauss-Seidel *inner* iterations on each block (see Section 2.2 and 4.1). The number of inner iterations was a tunable parameter of their tool. They analysed the tool by presenting results for a fixed and varying number of inner iterations.

Deavours and Sanders reported the solution of models with up to 15 million states on a workstation with 128MB RAM. Later, in [DS98a], they improved their earlier work by applying a compression technique on the matrix before storing it to disk, thus reducing the file I/O time. For our implementations of the explicit in-core and out-of-core methods, we have used the compact MSR scheme for matrix storage, which can also be considered as a compression technique. The compact MSR scheme provides 30% or more saving over conventional schemes. Furthermore, Deavours and Sanders report that the decompression time accounts for 50% of the computation time; therefore, in our case, we do not expect any overall gain from matrix compression.

Integer constant: B (*number of vector blocks*)
Semaphores: S_1, S_2: occupied
Shared variables: R_0, R_1 *(To read matrix A blocks into RAM)*
Shared variables: $Xbox_0$, $Xbox_1$ *(To read iteration vector x blocks into RAM)*

DiskIO 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 not an empty block
9.          read A_ij from disk
10.         if h ≠ 0
11.           read X_j from disk
12.       Signal(S_1)
13.       Wait(S_2)
14.       if h = 0
15.         write X_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, h
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.      X̃_i ← 0
7.      for h ← 0 to B − 1
8.        Wait(S_1)
9.        Signal(S_2)
10.       if j ≠ i
11.         if not an empty block
12.           X̃_i ← X̃_i − A_ij X_j
13.       else
14.         Solve A_ii X_i = X̃_i
15.         Test for convergence
16.       if j = 0 then j ← B − 1
17.       else  j ← j − 1
```

Fig. 8. A complete out-of-core block Gauss-Seidel iterative algorithm

The other notable papers on the matrix out-of-core solution are [KH99,BH01]. However, the main emphasis of these papers is on the parallelisation of the out-of-core approach of Deavours and Sanders which is not the subject of this paper.

4.4 A Complete Out-of-Core Approach

The limitation of the Deavours and Sanders' approach is the in-core storage of the iteration vector. In [KM02], Kwiatkowska and Mehmood extend the earlier out-of-core methods and present the *complete out-of-core* method which stores the matrix as well the iteration vector on disk. They solved the system $\pi Q = 0$ using the block Gauss-Seidel method.

We reformulate the system $\pi Q = 0$ as $Ax = 0$, as in Section 4.2, and give the complete out-of-core block Gauss-Seidel algorithm for the solution in Figure 8. The algorithm assumes that the vector and the matrix are divided into B and B^2 blocks of equal size, respectively. It also assumes that, before it commences, an initial approximation for the probability vector x has been stored on disk and that the approximation for the last block, X_{B-1}, is already in-core.

The complete out-of-core algorithm of Figure 8 uses two shared memory buffers, R_0 and R_1, to read blocks of matrix into RAM from disk (line 9). Similarly, vector blocks are read (line 11) from disk into shared memory buffers,

$Xbox_0$ and $Xbox_1$. Another array $\tilde{X}_i$, which is local to the *Compute Process*, is used to accumulate the sub-MVPs (line 12, *Compute Process*). As in Section 4.2, a local variable t can be used to switch between the pair of shared buffers, R_t and $Xbox_t$. Although the shared memory buffers are mentioned in the shared storage requirements of the algorithm, these buffers and the related local variables have been abstracted from the algorithm for the sake of simplicity.

The vector blocks corresponding to the empty matrix blocks are not read from disk (line 8, *DiskIO Process*). Moreover, to reduce disk I/O, the algorithm reads only the range of those elements in a vector block which are required for a sub-MVP. Once a vector block has been updated, this new approximation of the block must be updated on disk (line 15, *DiskIO Process*). The variable k (line 2, 18, *DiskIO Process*) is used to keep track of the index of the vector block to be written to disk during an inner iteration (line $7 - 17$, *DiskIO Process*). The variable j (line $5 - 6$ and $16 - 17$, *DiskIO Process*) is used to keep track of the indices of the matrix and vector blocks to be read from disk. The matrix blocks are stored on disk in such a way that a diagonal block (A_{ii}) follows all the off-diagonal blocks (all $A_{ij},\ 0 \leq j < B,\ j \neq i$) in a row of matrix blocks, implying that the diagonal block will always be read and used for computation when $h = B - 1$ (or when $j = i$). Similarly, the *Compute Process* uses the variable j (line $4 - 5$ and $16 - 17$) to ensure that all the off-diagonal sub-MVPs are accumulated (line 12) before the sub-system $A_{ii}X_i = \tilde{X}_i$ (line 14) is solved. In the light of this discussion, and the explanation of the matrix out-of-core algorithm given in Section 4.2, the *Compute Process* is self-explanatory.

Implementation

We have used two separate files to store the matrix (in compact MSR) and the iteration vector on disk. As for the matrix out-of-core solution, the matrix blocks for the complete out-of-core method have been stored on disk in an order which enables the *DiskIO Process* to read the file sequentially throughout an iteration. However, the case for reading through the file which keeps the vector is more involved because, in this case, the *DiskIO Process* has to skip those vector blocks which correspond to empty blocks of the matrix.

Finally, the complete out-of-core implementation uses an array of size B^2 to keep track of the zero and nonzero matrix blocks. A sparse scheme may also be used to store this information. The number of blocks B for the complete out-of-core solution is small, usually less than 100, and therefore the memory required for the array is negligible.

4.5 A Symbolic Out-of-Core Solution Method

In the complete out-of-core method, 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 while at the same time allowing for larger models to be analysed. This motivates the *symbolic out-of-core* method [KMNP02].

The idea of the symbolic out-of-core approach for the steady state solution of CTMCs 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. Once a vector block has been updated with the next approximation, it is written back to disk. The symbolic out-of-core solution uses offset-labelled MTBDDs [KNP02b] to store the matrix, while the iteration vector for numerical computation is kept on disk as an array. An improved implementation of the symbolic out-of-core method is reported in [Meh03].

Implementation
We have used a block iterative method to implement the symbolic out-of-core method. In this block method, we partition the system into a number of sub-systems and apply one (*inner*) iteration of the Jacobi iterative method[3] on each sub-system, in a *global* Gauss-Seidel iterative structure (see Section 2.2). This method is referred to in [Par02] as the *pseudo Gauss-Seidel method*.

The basic operation of the symbolic out-of-core block Gauss-Seidel method is the computation of the sub-MVPs, as was the case for the explicit out-of-core methods explained in earlier secions. The matrix is stored in an offset-labelled MTBDD. To implement each sub-MVP, we need to extract the matrix entries for a given matrix block from the MTBDD. We have seen in Section 3.3 that a matrix represented by an MTBDD can be decomposed into $(2^l)^2$ blocks by descending l levels of the MTBDD. Hence, in our implementation of the symbolic method, we select a value of l, take the number of blocks $B = 2^l$ 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. For large l, many of the matrix blocks may be empty. Therefore, instead of using a $B \times B$ array of pointers, a sparse scheme may be used to store the pointers to the nonzero matrix blocks; we select the compact MSR sparse scheme for this purpose. The extraction of matrix blocks, as required for each sub-MVP in the symbolic method, is therefore simple and fast; see [Meh03] for further details on this implementation.

5 Results

In this section, we compare performance of the out-of-core solution methods discussed in Section 4. We have implemented the algorithms on an UltraSPARC-II 440MHz CPU machine running SunOS 5.8 with 512MB RAM, and a 6GB local disk. We tested the implementations on three widely used benchmark models: a flexible manufacturing system (FMS) [CT93], a Kanban system [CT96] and a cyclic server Polling system [IT90]. These models were generated using PRISM

[3] The offset-labelled MTBDDs do not admit an efficient use of the Gauss-Seidel method, and therefore we use the Jacobi method to solve each sub-system.

[KNP02a], a probabilistic model checker developed at the University of Birmingham. More information about these models, a wide range of other PRISM case studies and the tool itself can be found at the PRISM web site [Pri].

The times to generate the files for the out-of-core solution phase are proportional to the times required to convert a model from BDD representation (see Section 3.3 on Page 9) to a sparse format. This file generation process can be optimised either for time or for memory. Optimising for memory can be achieved by allocating in RAM a data structure of the size of a submatrix of Q which is written to disk repeatedly as the conversion progresses. The generation process can be optimised for time by carrying out the entire process stated above in one step, i.e, converting the whole model into sparse format and then writing to file. We do not discuss the generation process any further, and hereon concentrate on the numerical solution phase.

The organisation of this section is as follows. In Section 5.1, we present and compare times per iteration for in-core and out-of-core versions for both explicit and symbolic approaches. In Section 5.2, we further analyse the out-of-core solutions with the help of performance graphs.

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

Table 2 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. The resulting number of reachable states are given in column 3. Column 4 lists the average number of off-diagonal entries per row, giving an indication of the sparsity of the matrices.

Columns 5–7 in Table 2 list the time per iteration results for "Explicit" implementations: these include the standard in-core, where the matrix and the vector are kept in RAM; the matrix out-of-core (see Section 4.2), where only the matrix is stored on disk and the vector is kept in RAM; and the complete out-of-core (Section 4.4), where both the matrix and the vector are stored on disk. We have used the Gauss-Seidel method for the three reported explicit implementations and the resulting number of iterations are reported in column 8. The matrices for the explicit methods are stored in the compact MSR scheme, which requires $4a + 3n$ bytes to store the matrix. The entries "a/n" in column 4 can be used to calculate the memory required to store the matrices.

Table 2 reports the time per iteration results for "Symbolic" implementations in columns 9–10: both in-core, where both the matrix and the iteration vector are stored in RAM, and out-of-core (Section 4.5), where the matrix is kept in RAM and the vector is stored on disk. The matrix for these symbolic implementations has been stored using the offset-labelled MTBDD data structure, and the vector is kept as an array. We have used a block iterative method for the symbolic implementations (see Section 4.5), and the respective number of iterations are reported in column 11. The run times for the FMS system ($k = 12, 13$) are taken for 50 iterations; we were unable to wait for their convergence, and hence the total numbers of iterations are not reported in the table.

Model	k	States (n)	a/n	Time (seconds per iteration)						
				Explicit				Symbolic		
				In-core	Out-of-core		Iter.	In-core	Out-of-core	Iter.
					Matrix	Complete				
FMS	6	537,768	7.8	0.3	0.5	1.1	812	1.0	1.3	916
	7	1,639,440	8.3	1.1	1.7	3.8	966	3.0	3.2	1,079
	8	4,459,455	8.6	3.2	5.1	10.7	1,125	8.9	10.4	1,245
	9	11,058,190	8.9	–	24	51.8	1,287	39.6	35.9	1,416
	10	25,397,658	9.2	–	69	146	1,454	149	142	1,591
	11	54,682,992	9.5	–	–	374	1,624	–	708	1,770
	12	111,414,940	9.7	–	–	–	–	–	1,554	>50
	13	216,427,680	9.9	–	–	–	–	–	3,428	>50
Kanban system	4	454,475	8.8	0.3	0.5	1.0	323	0.5	0.8	373
	5	2,546,432	9.6	1.8	3.0	6.0	461	3.1	4.5	532
	6	11,261,376	10.3	–	30	68.6	622	15.6	22.6	717
	7	41,644,800	10.8	–	180	283	802	–	143	924
	8	133,865,325	11.3	–	–	–	–	–	601	1,151
Polling system	15	737,280	8.3	0.5	0.7	1.2	32	0.8	0.8	263
	16	1,572,864	8.8	1.1	1.9	2.9	33	1.7	2.0	276
	17	3,342,336	9.3	2.4	3.9	6.4	34	3.9	4.6	289
	18	7,077,888	9.8	5.5	15.8	20.4	34	8.3	10.5	302
	19	14,942,208	10.3	–	41	71	35	20.3	23.8	315
	20	31,457,280	10.8	–	101	162	36	–	52	328
	21	66,060,288	11.3	–	–	359	36	–	177	340
	22	138,412,032	11.8	–	–	–	–	–	374	353

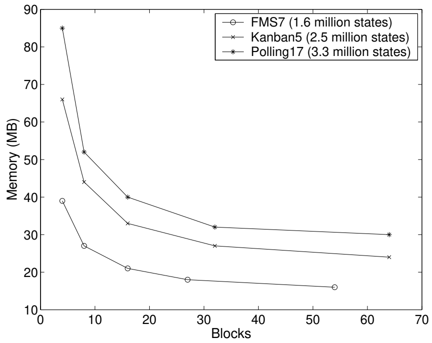
Table 2. Comparing times per iteration for in-core and out-of-core methods

The convergence criterion we have used in all our implementations is given by the equation (8) for $\varepsilon = 10^{-6}$. All reported run times are *wall clock* times.

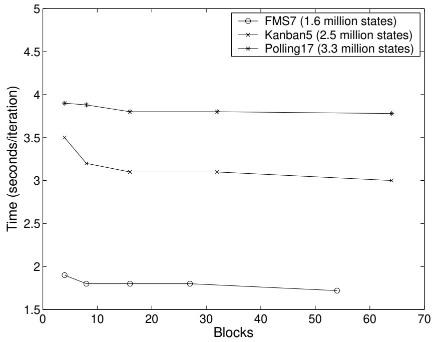
We note, in Table 2, that the in-core explicit method provides the fastest run-times. However, pursuing this approach, the largest model solvable on a 512MB workstation is the Polling system with 7 million states. The in-core symbolic solution can solve larger models because, in this case, the matrix is stored symbolically. The largest model solvable with this symbolic in-core approach is the FMS system with 25.3 million states. We now consider the out-of-core approaches. The matrix out-of-core solution requires in-core storage for one iteration vector and two blocks of the matrix. The memory required for these matrix blocks can, in general, be reduced by increasing the number of blocks. However, in this case, the largest solvable model is limited by the size of the iteration vector. Pursuing the matrix out-of-core approach, the largest model solvable on the workstation is the Kanban system with 41.6 million states.

The out-of-core storage of both matrix and vector can solve even larger models. This is reported in column 7, and the largest model reported in this case is the Polling system with 66 million states. The limit in this case is the size of the available disk (here 6GB). Finally, the largest solvable model on the available machine is attributed to the symbolic out-of-core approach, i.e., the FMS system with 216 million states. This is possible because, in this case, only the vector is stored on disk and the symbolic data structure provides a compact in-core representation for the matrix.

The matrix out-of-core and the symbolic in-core methods both provide a solution to the matrix storage problem, and hence it is interesting to compare the run times for the two approaches. We note in Table 2, for the Polling and Kanban systems, that the run times per iteration for the symbolic in-core are faster than



(a) Memory vs. blocks



(b) Time per iteration vs. blocks

Fig. 9. The matrix out-of-core method: effects of varying the number of blocks

the marix out-of-core method. However, for the FMS system, matrix out-of-core method provides better run times. Similarly, we note that the symbolic out-of-core method provides faster run times for Polling and Kanban systems, but is slower than the complete out-of-core approach for the FMS system.

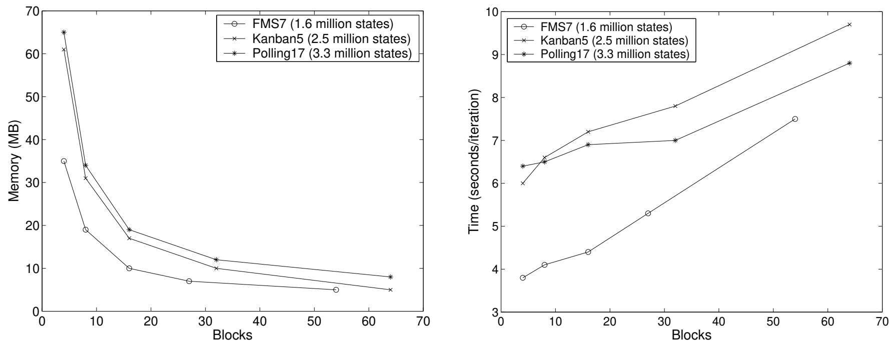
We observe that the results for explicit solutions are quite consistent for all three example models. However, the performance of symbolic solutions, both in-core and out-of-core, depends on the particular system under study. This is because the symbolic methods exploit model structure through sharing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

5.2 Further Analysis of the Solution Methods

In this section, we investigate performance for the out-of-core solution methods by analysing the memory and time properties plotted against the number of blocks. All experiments reported in this section have been performed on the same machine as in Section 5.1. We begin by analysing the matrix out-of-core solution and then move on to the complete and the symbolic out-of-core solutions.

The Matrix Out-of-Core Method

A matrix out-of-core algorithm was given in Section 4.2 and the time per iteration results presented in Table 2. In Figure 9(a), we have plotted the memory requirements of the matrix out-of-core solution for three CTMCs, one from each case study. The plots display the total amount of memory used against the number of vector blocks B of equal size ($B \times B$ matrix blocks). Consider the plot for the Polling system. The memory required to store the vector and matrix in this case, if kept completely in RAM, is approximately 26MB and 135MB



(a) Memory vs. blocks (b) Time per iteration vs. blocks

Fig. 10. The complete out-of-core method: effects of varying the number of blocks

respectively. Decomposing the matrix into blocks, keeping it on disk, and reading one block at a time reduces the total in-core memory requirement of the solution. The memory required for the case $B = 4$ is 85MB. Increasing the number of blocks up to $B = 64$ reduces the memory requirements to nearly 30MB. This minimum is bounded by the storage required for the iteration vector. Similar properties are evident in the plots for the FMS and Kanban system CTMCs.

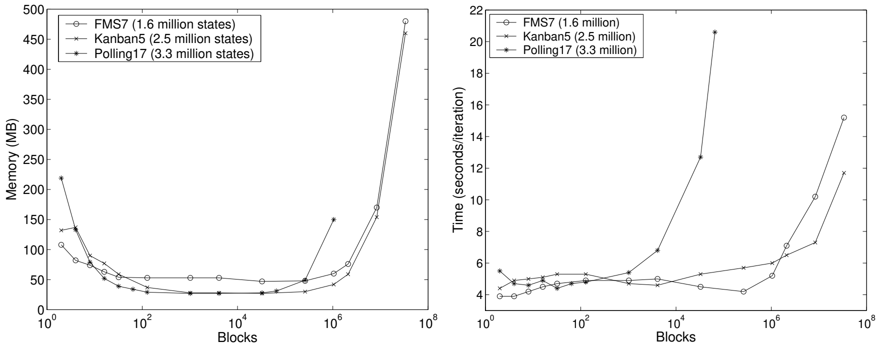
In Figure 9(b), we analyse the time per iteration characteristics for the same three CTMCs, plotted against the number of vector blocks. We note a slight decrease in the time per iteration for all three CTMCs. The reason for this slight decrease in time is the decrease in the memory requirement of the solution process, as demonstrated in the corresponding Figure 9(a). This effect would be more obvious for larger models. Another reason for this effect is that increasing the number of blocks typically results in smaller blocks, which possibly have less variations[4] in their sizes, consequently resulting in a better balance between the disk I/O and the computation. However, increasing the number of blocks to a very high number will possibly result in an increase in solution time due to the higher number of synchronisation points (see Section 4.2).

The Complete Out-of-Core Method

Figure 10(a) illustrates the total memory requirement of the complete out-of-core solution against the number of blocks for the same three CTMCs. The vector and the CTMC are partitioned into B and $B \times B$ blocks respectively, and, in this case, both are stored on disk. We note that the memory plots in the figure show a similar trend as in Figure 9(a).

The time per iteration properties of the complete out-of-core solution are plotted in Figure 10(b). In contrast to the matrix out-of-core method, we note a significant increase in solution time for the complete out-of-core method with an

[4] Although the blocks have an equal number of rows, they can still have a varying and unequal number of nonzero entries.



(a) Memory requirements vs. blocks (b) Time per iteration vs. blocks

Fig. 11. The symbolic out-of-core method: effects of varying the number of blocks

increase in the number of blocks. This is due to the fact that, for the complete out-of-core method, the iteration vector[5] must be read B times in each iteration. Consequently, an increase in the number of blocks generally results in an increase in the amount of disk I/O, and hence an increase in the solution time.

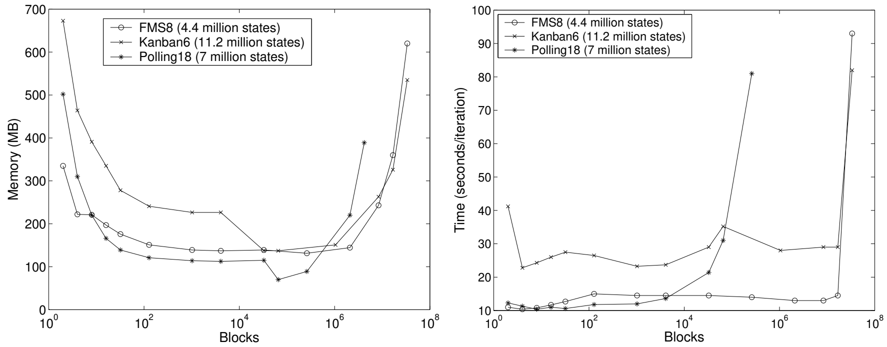
The Symbolic Out-of-Core Method

In Figure 11, we analyse performance of the symbolic out-of-core method by plotting the memory and time properties against the number of blocks for the same three CTMCs as in Figure 10. To further explore the behaviuor of the symbolic out-of-core method, in Figure 12, we plot similar properties for larger CTMCs. We note in the two figures that the range for the numbers of blocks is much higher compared to the earlier graphs for explicit solutions. The reason is that the MTBDD actually encodes a matrix over its *potential state space*, which typically includes many unreachable states (see Section 3.3). In the following, we explain the plots for Figure 12; in the light of this discussion, Figure 11 will be self-explanatory.

The total memory requirement of the symbolic out-of-core solution against the number of blocks for three CTMCs is plotted in Figure 12(a) (compare with Figure 11(a)). We explain the plot for the Kanban system ($k = 6$). The memory required for the case $B = 2$ is above 650MB. The increase in the number of blocks reduces the memory requirements for the Kanban system to nearly 140MB. A similar properties are evident for the other plots in Figure 12(a). For large numbers of blocks (i.e. the rightmost portions of the plots), we note an increase in the amount 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.

The time per iteration properties of the symbolic out-of-core solution are analysed in Figure 12(b), plotted against the number of vector blocks. Consider

[5] Of course, the vector blocks corresponding to the zero matrix blocks are not read.



(a) Memory requirements vs. blocks

(b) Time per iteration vs. blocks

Fig. 12. The symbolic out-of-core method: effects of varying the number of blocks

the plot for the Kanban system. Initially, for $B = 2$, the memory required (see Figure 12(a)) for the iteration vector 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 rationale for this is as follows. As we explained in Section 3.3, our decomposition of the MTBDD matrix into blocks can result in a partitioning such that the resulting matrix (and hence vector) blocks are of unequal 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 12(a). Note that, for the symbolic implementations, we use a block iterative method where we apply one Jacobi (inner) iteration on each sub-system, in a global Gauss-Seidel iterative structure (see Section 4.5). Increasing the number of blocks, therefore, typically causes a reduction in the required number of iterations for a CTMC to converge.

We conclude this section here with the observation that among the many factors which affect performance of the out-of-core solutions are the data structures that hold the matrix, and the number of blocks, B, that the matrix (and vector) are partitioned into. We have also found that, under different values of k (see Table 2) for each case study, similar patterns for the memory and time plots against the number of blocks are observed. This can, in fact, be useful for predicting good choices of B for larger values of k. A useful direction for future work would be to investigate more fully what constitutes a good choice for the number of blocks.

6 Conclusions

In this paper, serial out-of-core algorithms for the analysis of large Markov chains have been surveyed. Earlier work, in this context, has focussed on implicit and parallel explicit solutions. During the last five or so years, some progress has been made, beginning with the solution of a 15 million states system on a workstation with 128MB RAM, with no assumption on its structure [DS97,DS98a]. This approach was parallelised in 1999 [KH99], leading to the solution of 724 million states on a 26-node dual-processor cluster [BH01]. A number of solution techniques had been devised by the late 1990s to cope with the matrix storage problems. However, explicit storage of the solution vector(s) hindered further progress for both implicit and explicit methods. These limitations were later relaxed by the out-of-core storage of the vector. The complete out-of-core solution of a 41 million states system on a 128MB RAM machine was demonstrated in [KM02]. Furthermore, a combination of MTBDD-based in-core symbolic storage of matrix and out-of-core storage of the vector allowed the solution of a 216 million states system on a single workstation with 512MB RAM [KMNP02,Meh03].

We surveyed the out-of-core approaches in this paper. The algorithms were analyzed using tabular and graphical results of their implementations with the help of benchmark case studies. Our focus in this paper has been the steady state solution of an (irreducible) CTMC. We intend to generalise these techniques to other numerical computation problems, such as transient analysis of CTMCs and analysis of DTMCs and MDPs. Another direction for future research is to extend the complete out-of-core and the symbolic out-of-core approaches by employing parallelisation.

The Kronecker approach provides a space-efficient representation of a Markov chain. Representations based on such an approach have increasingly gained popularity. These methods, however, still require explicit storage of the solution vector(s). The out-of-core storage of the solution vector can also provide a solution in this case.

Further improvements in out-of-core techniques can be achieved with the help of redundant arrays of independent disks (RAID). In future, we anticipate that a combination of parallel and out-of-core techniques will play an important role in the analysis of large stochastic models.

Acknowledgment

I am grateful to Markus Siegle and Alexander Bell for their extremely valuable suggestions and many fruitful discussions during the write up of this manuscript. David Parker, Gethin Norman and Marta Kwiatkowska are all very kindly acknowledged for their support and helpful comments.

References

[Bau93] F. Bause. Queueing Petri Nets: A Formalism for the Combined Qualitative and Quantitative Analysis of Systems. In *Proc. PNPM'93*. IEEE Computer Society Press, October 1993.

[BBC+94] 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.

[BCMP75] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Journal of ACM*, 22(2), 1975.

[Bel99] A. Bell. Verteilte Bewertung Stochastischer Petrinetze, Diploma thesis, RWTH, Aachen, Department of Computer Science, March 1999.

[BFG+93] 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.

[BG96] M. Bernardo and R. Gorrieri. Extended Markovian Process Algebra. In *Proc. CONCUR 1996, LNCS Volume 1119*, Italy, 1996. Springer-Verlag.

[BH01] 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.

[BK03] P. Buchholz and P. Kemper. Kronecker based Matrix Representations for Large Markov Models. In *this Proceedings*, 2003.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, May 1993.

[CG89] A. E. Conway and N. D. Georganas. *Queueing Networks - Exact Computational Algorithms: A Unified Theory based on Decomposition and Aggregation*. MIT Press, 1989.

[CM99] G. Ciardo and A. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proc. PNPM'99*, Zaragoza, 1999.

[CT93] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

[CT96] 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.

[DS97] D. D. Deavours and W. H. Sanders. An Efficient Disk-based Tool for Solving Very Large Markov Models. In *Proc. TOOLS'97*, volume 1245 of *LNCS*, pages 58–71. Springer-Verlag, 1997.

[DS98a] D. D. Deavours and W. H. Sanders. An Efficient Disk-based Tool for Solving Large Markov Models. *Performance Evaluation*, 33(1):67–84, 1998.

[DS98b] 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.

[Her92] M. Heroux. A proposal for a sparse BLAS Toolkit, Technical Report TR/PA/92/90, Cray Research, Inc., USA, December 1992.

[Hil94] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. NSMC'99*, Zaragoza, 1999.

[HR94] H. Hermanns and M. Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In *Proc. PAPM'94*, Germany, 1994.

[IT90] O. Ibe and K. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[Kah58] W. Kahan. *Gauss-Seidel methods of solving large systems of linear equations*. PhD thesis, University of Toronto, 1958.

[KH99] W. J. Knottenbelt and P. G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In *Proc. NSMC'99*, 1999.

[KM02] 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*.

[KMNP02] 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`).

[KNP02a] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. TOOLS'02*, volume 2324 of *LNCS*, 2002.

[KNP02b] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *Proc. TACAS 2002, LNCS Volume 2280*, April 2002.

[MBC84] M. A. Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2), 1984.

[MBC+95] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, and D. Kartson. *Modelling With Generalized Stochastic Petri Nets*. John Wiley & Son Ltd, 1995.

[Meh03] R. Mehmood. *On the Development of Techniques for the Analysis of Large Markov Models*. PhD thesis, University of Birmingham, 2003. To appear.

[Mol82] M. K. Molloy. Performance Analysis using Stochastic Petri Nets. *IEEE Trans. Comput.*, 31:913–917, September 1982.

[MP03] A. Miner and D. Parker. Symbolic Representations and Analysis of Large State Spaces. In *this Proceedings*, 2003.

[PA91] B. Plateau and K. Atif. Stochastic Automata Network for Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10), 1991.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, August 2002.

[Pla85] 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*, 1985.

[Pri] PRISM Web Page. `http://www.cs.bham.ac.uk/~dxp/prism/`.

[Saa90] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, NASA Ames Research Center, CA, 1990.

[Sie01] M. Siegle. Advances in Model Representations. In *Proc. PAPM/PROBMIV 2001, LNCS Volume 2165*, Aachen, Germany, 2001. Springer Verlag.

[Ste94] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[Tol99] S. Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.