Out-of-Core Solution of Large Linear Systems of Equations arising from Stochastic Modelling

Marta Kwiatkowska and Rashid Mehmood

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

Abstract. Many physical or computer systems can be modelled as Markov chains. A range of solution techniques exist to address the state-space explosion problem, encountered while analysing such Markov models. However, numerical solution of these Markov chains is impeded by the need to store the probability vector(s) explicitly in the main memory. In this paper, we extend the earlier out-of-core methods for the numerical solution of large Markov chains and introduce an algorithm which uses a disk to hold the probability vector as well as the matrix. We give experimental results of the implementation of the algorithm for a Kanban manufacturing system and a flexible manufacturing system. Next, we describe how the algorithm can be modified to exploit sparsity structure of a model, leading to better performance. We discuss two models, a cyclic server polling system and a workstation cluster system, in this context and present results for the polling models. We also introduce a new sparse matrix storage format which can provide 30% or more saving over conventional schemes.

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 a system as a Continuous Time Markov Chain (CTMC), provided the probability distribution is restricted 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. Hence, the problem of analysing Markov models 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 so-called state-explosion (also known as largeness) problem in this area. These, among others, include symbolic techniques [6, 12, 16, 25], Kronecker methods [23], and *on-the-fly* methods [9]. However, the solution processes for these methods have high time complexity. While some new developments such as Matrix Diagrams (MDs) [2, 5, 22] and hybrid MTBDD methods [21] can allow a very compact and time efficient encoding of the 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 some 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 to store the CTMC matrices [7, 8] and parallelising the disk-based numerical solutions [19, 1]. However, the memory requirements for these methods are also dominated by the storage of a vector of size proportional to the number of states in the model.

This paper makes two novel contributions to the numerical solution of Markov chains. The first contribution is an extended out-of-core algorithm which relaxes the memory limitations for the explicit numerical solution methods caused by the need to store the probability vector(s) explicitly. This approach is also applicable to implicit methods, see §6. We use the term "implicit methods" in this paper for the numerical solution methods which use some kind of symbolic data structure (MTBDDs, MDs, etc) 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 (Modified Sparse Row etc). 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].

The algorithm introduced in this paper assumes no structure in the sparse matrix, and hence can be applied to any CTMC model. The algorithm divides the underlying CTMC matrix and the probability vector into a certain number of blocks and stores these blocks onto a disk. It reads into main memory a block of the matrix and of the probability vector, performs the required calculation and writes back updated elements of the probability vector onto the disk. Moreover, the algorithm provides fairly fast convergence due to the Gauss-Seidel iterative method being employed. To obtain high 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 that the matrix and the probability vector is divided into, the less memory is required by the algorithm. Virtually, it leaves no upper bound on the size of a numerically solvable CTMC matrix on a standard workstation provided sufficient disk space is available. However, there is an obvious tradeoff between the memory requirement of the algorithm and the resulting solution time. We demonstrate the behaviour of the algorithm by implementing it for an UltraSPARC-II 360MHz machine with 128MB physical memory (RAM) and a 6GB local disk. We test the implementation on a Kanban manufacturing system [3] and a Flexible Manufacturing System (FMS) [4] and present results.

The performance of the algorithm can be improved if a certain sparsity structure can be found in the underlying CTMC matrix. This fact is demonstrated by including results of a modified algorithm applied to a Cyclic Server Polling System [17].

The second contribution of this paper is a new sparse matrix storage format, called the *Compact Modified Sparse Row* (CMSR) or *compact MSR* format. We believe that this storage format, which can also be used with in-core iterative solutions of sparse linear systems, would be helpful in further pushing the upper limit on the size of solvable models in the area of Markov modelling. We have used the compact MSR format in the implementations of various serial and parallel algorithms, including the algorithm presented in this paper.

The Tool PRISM. All the CTMC matrices used in this paper are generated by the PRISM tool [21, 20]. PRISM, a probabilistic model checker, is a tool for analysing probabilistic systems currently being developed at the University of Birmingham. It supports three models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. For the numerical solution phase, it provides three choices: one using pure MTBDDs, one based on in-core sparse iterative methods, and a third using a hybrid approach. Several case studies have been performed, see `http://www.cs.bham.ac.uk/~dxp/prism/` for details. This paper is part of an effort to improve the range of solution methods provided by PRISM.

2 Numerical Methods

We focus in this paper on numerical solutions 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 state transition rate matrix, 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 $Ax = b$, provided $A = Q^T$, $x = \pi^T$, and $b = 0$, can be used. For a discrete-time Markov chain, the problem of model checking a probabilistic *until* formulae [13] reduces to solving $Ax = b$ where A is a sub-matrix of the transition probability $n \times n$ matrix P and b is a vector of the appropriate size.

The numerical solution methods for linear systems of the form $Ax = b$ are broadly classified into direct methods and the 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 convergence is achieved. See [11] for further information on direct methods and [24, 26] for iterative methods.

We concentrate here on the Gauss-Seidel (GS) iterative method, which we employed for our out-of-core algorithm. In each iteration of the Gauss-Seidel algorithm we calculate

$$\pi_i^{(k)} = -\frac{1}{q_{ii}} \left(\sum_{j=0}^{i-1} \pi_j^{(k)} q_{ij} \; + \sum_{j=i+1}^{n-1} \pi_j^{(k-1)} q_{ij} \right), \tag{2}$$

for $0 \leq i < n$. Unlike Jacobi iteration, each component of the solution vector in the Gauss-Seidel method must be updated sequentially. However, Gauss-Seidel has advantages over Jacobi in that it has a faster convergence rate and that it requires only one vector of size n. We chose the relative measure

$$\max_i \left(\frac{\mid \pi_i^{(k)} - \pi_i^{(k-1)} \mid}{\mid \pi_i^{(k)} \mid} \right) \; < \; \epsilon, \tag{3}$$

for some ϵ $(0 < \epsilon \ll 1)$ to test for convergence in all the experiments presented in this paper.

3 Sparse Storage Considerations

Iterative solution of a system of linear equations of the form $\pi Q = 0$ involves explicit storage of the whole transition rate matrix Q. A usual choice for the data structure is the *modified sparse row* (MSR) format [24]. The MSR format stores all the off-diagonal non-zero entries in an array, say *Val*, of floats (4 bytes) or doubles (8 bytes). The corresponding column indices are stored in a separate array of integers (4 bytes), say *Col*. Diagonal entries may be stored in the first or the last n locations of the array *Val*. Another option is to store the n diagonal entries in a separate array. Further, an array of $n + 1$ integers is used, the ith entry of which points to the first entry of the ith row in the arrays *Val* and *Col*. The total memory requirements, if non-zeros are stored as floats, is $8(a+n)$ bytes, and is $12(a+n)$ bytes when doubles are used to store the non-zero entries; a is the number of the off-diagonal non-zero entries in the matrix. The *modified sparse column* (MSC) storage format differs from MSR in that it stores the matrix entries and their positions by column.

Table 1 lists some statistics about the matrices taken from a case study based on a Cyclic Server Polling system [17], obtained with the help of the tool PRISM. The table also lists the memory requirements to store the CTMC matrix and the solution vector for each model. Column 1 lists the parameter *Poll*, each value representing a separate CTMC model, and column 2 shows the resulting number of states n which equals the dimension of the matrix. Column 3 lists the number of the off-diagonal non-zero entries a. Column 4 gives the average off-diagonal non-zero entries per row, an indication of the sparsity of the matrices.

The number of distinct values in the CTMC matrices resulting from the models we considered is relatively small. This important characteristic can lead to appreciable memory savings. The distinct values may be stored in a small array, and the index to these values can be stored as a short (2 bytes). Column indices may be stored in a separate array of integers (4 bytes). The number representing the count of the off-diagonal non-zero entries in a row can be stored as an unsigned char (1 byte). The storage of the diagonal will require 8 bytes (doubles) per each entry. The total memory requirements for this scheme is $6a+9n$ bytes plus the storage for the actual distinct values of the matrix nonzero entries. Since the storage for the actual distinct values is relatively small for large models (in comparison with the total memory requirement), we do not consider it in future discussions.

Let us mention here a possible transformation of the system $\pi Q = 0$ into an equivalent system $R^T y^T = 0$ [1]. If we define $R = -QD^{-1}$, then the system $\pi Q = 0$ can be equivalently written as $\pi DQD^{-1} = -yR = 0$, with $y = -\pi D$. The matrix D is defined as the diagonal matrix with $d_{ii} = q_{ii}$ for $0 \leq i < n$. Effectively, we solve the equivalent system $R^T y^T = 0$, with all the diagonal entries of the matrix R being -1. Along the lines of the storage scheme $(6a+9n)$ described in the previous paragraph, the storage requirements for this system is $6a + n$ with an additional advantage of runtime savings of n divisions per each iteration. For future references (in this paper) we call this memory scheme the **distinct-once** memory scheme, implying that the distinct values in the matrix are stored only once. This storage format is currently being used in iterative solutions for the explicit storage of CTMC matrices [1, 7, 19].

3.1 A Compact MSR Sparse Storage Format

This section introduces and explains a new storage scheme that renders a 30% or more reduction over the distinct-once storage scheme. The scheme, used to store the sparse CTMC matrix, preserves the general applicability of the explicit iterative solutions of Markov chains, whether in-core, out-of-core or parallel. Moreover, it does not increase the runtime of the solution process; indeed, it

Poll	*states* (n)	*off-diagonal nonzero* (a)	a/n	MB required for Q		MB per π
				MSR	*distinct-once*	
17	3,342,336	31,195,136	9.33	260	198	26
18	7,077,888	69,599,232	9.83	584	438	54
20	31,457,280	340,787,200	10.83	2839	2129	240
21	66,060,288	748,683,264	11.33	6323	4421	513

Table 1. Storage requirements in Mbytes for the Polling models

reduces the runtime due to better caching effect. We call it the *Compact Modified Sparse Row* or *compact MSR* storage format. A similar *compact MSC* is possible.

In the distinct-once scheme, which is described earlier in the parent section (§3), the distinct values are stored once in an array of doubles and the indices to these values are stored in another array of shorts. In total, this scheme requires $6a + n$ bytes if matrix R is stored, and requires additional $8n$ bytes for diagonal elements if matrix Q is stored. The number of distinct values in the matrix R for various models are around ten thousand to twenty thousand [1, 19], hence the indices are stored as shorts.

We observe that the operation QD^{-1} increases the number of distinct values in Q and the number of distinct values in D. The number of distinct values to be stored may be decreased by storing the matrix Q instead of storing the matrix $R = QD^{-1}$. Furthermore, diagonal elements also have relatively few distinct values; the same philosophy of distinct values for transition rate matrix can be applied to the diagonal vector. The distinct values in the diagonal elements may be stored as *1/value* instead of *value*. The storage of diagonal vector would require n or $2n$ bytes depending on the maximum possible number of distinct values.

Among the various models used in the literature, probably FMS models [4] have the highest number of distinct values [1, 19]; they are a good example to demonstrate the effectiveness of the compact MSR storage scheme. Table 2 summarises a fact sheet for FMS models comparing storage requirements in MSR, distinct-once, and compact MSR formats. The model *fms10* has 25.4 million states, contains 125 (not given in the table) distinct values in the off-diagonal elements of matrix Q, and 1157 (not given in the table) distinct values in the diagonal elements of matrix Q. Counting up to the index "125" requires 7 binary bits of memory. A data type int uses 32 bits of storage. If we steal 7 bits from the allocated space for an unsigned int, we would still have 25 bits; these are sufficient to address up to 33.5 million states. The diagonal elements require additional $2n$ bytes for the storage of n short indices to an array of doubles, which stores the distinct values in the diagonal vector. The total memory requirements in the compact MSR format for FMS models is thus $4a + 3n$ bytes.

The proposed scheme of bitwise packing of the index of distinct values together with a state index within an int variable is space efficient but subject to a trade-off between the number of distinct elements and the model size. For example, the model *Poll20*, which has 31.45 million states (see Table 1), contains just

fms	states	nonzero	memory required for Q (MB)		
		(*off-diagonal*)	*MSR format*	*distinct-once*	*compact MSR*
8	4,459,455	38,533,968	326	225	160
10	25,397,658	234,523,289	1975	1366	967
12	111,414,940	1,078,917,632	9042	6279	4433

Table 2. Storage requirements for the FMS models in various formats

3 (not given in the table) distinct values in the off-diagonal elements of matrix Q, and 77 distinct values in the diagonal elements of matrix Q. Since only 2 bits of memory are required to index off-diagonal distinct values, this allows us to address models of up to one billion states. The compact MSR format for the Polling models requires $4a + 2n$ bytes. The limitation on the maximum number of states is not a serious concern for parallel and disk-based solutions, where local indices within a matrix block are typically used. Out-of-core solution will also benefit through a reduction in disk access time.

We now turn our attention to performance issues of the scheme, which will also help explain the implementation issues. We know that modern microprocessors have several functional units which are pipelined to deliver performance in parallel. These independent units help in overlapping the execution of instructions when the instructions are independent of each other. This potential overlap among instructions is called **instruction-level parallelism** (ILP) since the instructions can be evaluated in parallel [15]. The availability of several functional units enables parallel execution of mutually independent instructions. Encoding (compact MSR format) the data (indices to distinct values and their positions in matrix) into a 4 byte structure (unsigned int) may be carried out using bitwise operations. Most modern processors have two simple integer units which can execute this type of operations in a single clock cycle. If the instructions in the solution process are arranged in a manner that allows instruction-level parallelism to be exploited by the microprocessor, the additional operations added due to the compact MSR format do not slow down the code.

We have described the compact MSR format here, which offers a 30% memory savings. The savings increase with the increase in the value of a/n. The average number of nonzero elements per row (a/n) in a matrix normally increases with the increase in the size of the model. It can be calculated that for $a/n = 11$, the new scheme provides 32.7% savings for FMS models and 34.4% savings for the Polling models over the distinct-once format. All performance related claims about the compact MSR format in this section are made after implementations and careful analysis of the storage scheme.

Serial Solutions. Serial out-of-core algorithms benefit from compact MSR in two ways: by reducing the main memory and file space requirements, and by reducing the amount of disk I/O. Moreover, dividing the CTMC matrix into blocks (see §4) with local state numbering relaxes the limits on the largest model solvable by serial iterative solutions. We use the term "*serial out-of-core*" to refer to a solution where one process is used to do major computations and another process is used to schedule disk I/O.

Parallel Solutions. In-core parallel and parallel out-of-core solutions would both benefit from the compact MSR format. Local state numbering for all blocks, as is the case in parallel solutions, would relax the limits on the largest models solvable using this format.

4 A Complete Out-of-Core Solution

The idea of using disk to store data structures is motivated by the problems associated with the very high memory requirements of the numerical solution process. Schemes such as MTBDDs [6], Kronecker methods [23, 10], MDs [2, 22, 5] and the hybrid MTBDD approach of [21] provide a way of storing a CTMC matrix in a compact form. However, obtaining high performance from these methods is dependent on a certain structure in the model.

An alternative approach in stochastic modelling, introduced by Deavours and Sanders [7, 8], is to use a *disk-based* (out-of-core) solution technique. They claimed, by comparing their tool with Kronecker [23] and "on-the-fly" [9] methods, that a disk-based approach is the method of choice for solving large Markov models provided sufficient disk space is available. Further work on the parallelisation of out-of-core methods in [18, 19, 1] has justified their claim.

A limitation common to the symbolic methods and to the serial and parallel out-of-core methods is the need to store one or more solution vectors of size n, the number of states in the CTMC model. Table 1, in its last column, lists the storage requirements for a probability vector. In this section we present an algorithm which relaxes these limits. The algorithm assumes no structure in the underlying CTMC matrix. The idea behind the algorithm is to store the CTMC matrix and the probability vector onto the disk, read a subset of the vector and of the matrix into RAM, perform computations, update the approximations to the probability vector, and write back the new approximation onto disk. Concurrency and synchrony play an important role in the performance of the algorithm.

4.1 The Algorithm

The out-of-core algorithm, given in Figure 1, comprises two concurrent processes: the *Disk-IO Process* and the *Compute Process*. Performance of the algorithm relies mainly on synchronised execution of the two processes; these issues are discussed in §4.2. For now, let us assume that the two processes in Figure 1 execute concurrently and synchronously; they synchronise by calling the functions *Wait()* and *Signal()*.

The algorithm assumes that the CTMC matrix to be solved is stored in a file on disk. The algorithm does not make a distinction between the storage format used. However, we have used the compact MSR format in our implementations. The CTMC matrix is divided into $B \times B$ square blocks of dimension $n/B \times n/B$, where some of the blocks might be empty. The algorithm assumes $N \ mod \ B = 0$. Figure 2 shows the sparsity pattern of a matrix resulting from a Flexible Manufacturing System (FMS) model [4]. The figure also explains the division of the matrix into 4×4 blocks. These blocks are numbered Q_{ij} for $0 \leq i, j < B$. The matrix blocks are stored in a file in such a way that all the row blocks – all $(i, 0)$ to $(i, 3)$ for row i in Figure 2 – are adjacent to each other. The aforementioned tool PRISM (§1) has been adapted to generate files according to these requirements.

Integer constant: B (*number of blocks*)
Semaphores: S_1, S_2: occupied
Shared variables: R_0, R_1 (*To read matrix Q 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. $k \leftarrow B - 1$
3. **while** not converged
4.   **for** $i \leftarrow 0$ to $B - 1$
5.   **if** $i = 0$ **then** $j \leftarrow B - 1$
6.   **else** $j \leftarrow i - 1$
7.   **for** $h \leftarrow 0$ to $B - 1$
8.   **if** not an *empty* block
9.   read Q_{ij} from disk
10.   **if** $h \neq 0$
11.   read Π_j from disk
12.   Signal(S_1)
13.   Wait(S_2)
14.   **if** $h = 0$
15.   write Π_k to disk
16. $k \leftarrow k + 1 \ \ mod\ B$
17.   **if** $j = 0$ **then** $j \leftarrow B - 1$
18.   **else** $j \leftarrow j - 1$

Compute Process

1. Local variable: i, j
2. **while** not converged
3.   **for** $i \leftarrow 0$ to $B - 1$
4.   **if** $i = 0$ **then** $j \leftarrow B - 1$
5.   **else** $j \leftarrow i - 1$
6.   **for** 0 to $B - 1$
7.   Wait(S_1)
8.   Signal(S_2)
9.   **if** $j \neq i$
10.   **if** not an *empty* block
11.   Accumulate $Q_{ij}\Pi_j$
12.   **else**
13.   Update Π_i
14.   check for convergence
15.   **if** $j = 0$ **then** $j \leftarrow B - 1$
16.   **else** $j \leftarrow j - 1$

Fig. 1. An *out-of-core* Gauss-Seidel iterative algorithm

The probability vector π is also divided into B blocks or subsets, 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 subset Π_i. The algorithm also assumes that the initial approximation for all the blocks of the probability vector π is stored on the disk except for the last block Π_{B-1}.

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 Gauss-Seidel method, which was described in §2. We stress that we are using the point version of Gauss-Seidel iterative method, as opposed to the block version which applies the computations given by equation (2) at block level. The matrix Q and the vector π are read into RAM one block after another. Therefore, in order to calculate the next approximation for the subset Π_i, we need to accumulate the products $Q_{ij}\Pi_j$ in an array of *doubles* of size n/B for all $0 \leq j < B$ save $j \neq i$. These products are accumulated by line 11 of the Compute Process in a local array. The shared

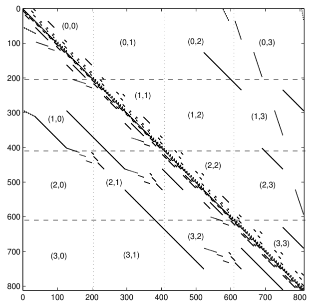

Fig. 2. Sparsity structure of FMS CTMC matrices

memory blocks R_x and Πbox_x, two each, are used to hold matrix and probability vector blocks respectively. All the elements of the subset Π_i are updated using equation (2), during the last iteration ($j = i$) of the inner loop (lines $6 - 16$). The convergence test, as given by equation (3), is carried out before updating each element of the subset Π_i. Before proceeding to calculations in each iteration of the inner loop, the Compute Process waits for the signal from the Disk-IO Process – an indication that the Disk-IO Process has loaded the shared memory blocks with the new data.

Disk-IO Process. The Disk-IO Process is responsible for reading the matrix and the probability vector blocks from disk into RAM, and for writing the new approximations of the probability vector onto disk. The lines $5 - 6$ along with lines $17 - 18$ of the Disk-IO Process determine the indices of the matrix block and the probability vector block to be read during an inner loop (lines $7 - 18$). The process reads a matrix block and a probability vector block in each iteration of inner loop and signals the Compute Process. It proceeds to the next iteration of the inner loop if the block is empty. To reduce the amount of I/O, only the range of the required elements of the probability vector are loaded from disk (except during the last iteration of the inner loop). For this purpose, we keep track of the lowest and the highest index of the required elements of a particular matrix block. During the first iteration of the inner loop, the newly calculated subset of the probability vector is written to disk, except otherwise when blocks of vector π are loaded from disk into the main memory.

4.2 Implementation

The design of an out-of-core numerical solver involves concurrency, and hence requires communication and synchronisation. We have two options available in UNIX: multithreaded code or multiprocess code. Threads in a process share address space and thus provide a convenient way of communication. Processes in UNIX are also allowed to do interprocess communication (IPC).

Model	l	states	blocks	file size (MB)	RAM (MB)	*Time1* (sec/it)	*Time2* (sec/it)	Iter.	MB per π
Kanban	5	2,546,432	32	119	9	9	7	214	20
	6	11,261,376	64	575	20	90	69	289	86
	7	41,644,800	48	2164	105	561	308	374	317
FMS	8	4,459,455	27	195	16	34	21	893	34
	9	11,058,190	22	500	50	82	57	1018	84
	9	11,058,190	55	570	24	105	67	1018	84
	10	25,397,658	54	1266	54	280	178	1146	194

Table 3. Numerical solution times for the FMS and the Kanban models

We have implemented our out-of-core algorithm using two separate processes communicating via shared memory and synchronising using semaphores. In UNIX, the shared memory facility normally provides the most efficient interprocess communication of all the IPC mechanisms. It allows more than one process at a time to attach a segment of physical memory to its virtual address space. The communication among the processes is made possible by allowing the write access for more than one process. The semaphore mechanism can be used for synchronising the processes and to prevent inconsistencies. Our out-of-core algorithm achieves communication between the two processes by using the shared memory blocks R_x and Πbox_x, and synchronises through two semaphores, S_1 and S_2.

Generating Files. It has already been stated that the files for the out-of-core algorithm (in its required format) are generated by the tool PRISM. The times for the generation process are proportional to the times required to convert a model from BDD to a sparse format. Memory requirement for the file generation process can be either optimised for time or for memory. Optimising for memory can be achieved by keeping in main memory the data structures of the size of a small submatrix of the underlying CTMC matrix Q, and writing them onto file when the conversion is done; the process is repeated until the whole matrix is converted and written to the file. The generation process can be optimised for time by carrying out whole of the above process in one step, i.e, converting the whole model into sparse format in one step and then writing to file. Obviously, for efficiency reasons binary I/O is used. However, our concern in this paper is numerical solution of large linear systems, for the out-of-core numerical algorithm is independent of the modelling tool used to generate the data file.

4.3 Results

We tested the implemented out-of-core algorithm on four large models available in the literature. These are: a Cyclic Server Polling System [17], a Kanban System [3], a Workstation Cluster [14], and a Flexible Manufacturing System [4]. The Polling and the Cluster models have sparsity structures which can be exploited

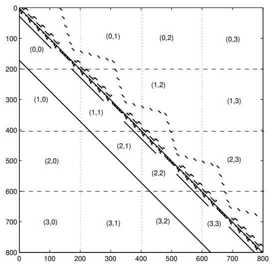

Fig. 3. Sparsity structure of matrices for the Cluster models

to obtain better performance; they are discussed in §5. In Table 3, we present solution times for FMS and Kanban CTMC matrices along with the memory requirements and the number of iterations the models required to converge. The convergence was tested according to equation (3) for $\epsilon = 10^{-6}$. The results were taken on an UltraSPARC-II 360MHz machine with 128MB RAM and a 6GB local disk; *Time1* in column 7 reports these results. All the solution times reported are real times.

Column 4 in Table 3 lists the number of blocks the matrix of a particular model is partitioned into and column 6 lists the corresponding amount of memory used. Column 5 gives the sizes of the files that store the complete matrices on disk. The last column gives an idea of the amount of memory required if the probability vector had to be kept in main memory; for *kanban7* and *FMS10*, this is simply not possible (we do not consider virtual memory). It is clear that the memory requirement for a particular model can be further reduced by increasing the number of blocks. However, this may lead to an increase of the solution time. This fact is demonstrated by listing the results for the *FMS9* model with two different numbers of block. Finally, we stress that the results presented in the table are for the out-of-core method where both the matrix and the probability vector are stored on disk, though for models smaller than $14-15$ million states the vector can be stored in main memory and faster run time may be obtained.

Among the recent developments concerning implicit methods, we would like to compare the results of our algorithm with the work on MDs presented in [5, 22] and with the work on the hybrid MTBDD method presented in [21]. For comparison purposes we have also executed our out-of-core algorithm on a workstation with an UltraSPARC-II 440MHz CPU; column 8 (*Time2*) in Table 3 lists these results. The algorithm in [5] requires more memory, for it needs to keep a vector of n doubles (the number of states) in the main memory. However, despite the higher memory requirements, the run times per iteration presented in [5] are considerably slower than our results.

The other approach we consider for comparison is the hybrid data structure [21] where MTBDDs are used to store the matrix, and the vector is stored explicitly. The results (run times per iteration) presented in [21] for the Kanban models are considerably better than ours. Their results presented for smaller

FMS models[1] are also better than ours. The differences between their per iteration run time results and ours become less pronounced for larger FMS models. However, the critical factor is that the hybrid method employs JOR or Power methods which requires two solution vectors and have, for the models concerned, considerably slower convergence rate than Gauss-Seidel. Finally, we note that the hybrid method for the *FMS10* model requires at least 388MB ($2 \times n$ doubles) using Jacobi, while our Gauss-Seidel out-of-core algorithm with 54MB RAM delivers faster run times and better convergence rate.

Finally, we stress that the out-of-core algorithm introduced in this paper is an attempt to relax the memory limitations in the analysis of Markov models. It enables one to extend the range of solvable Markov models on a standard workstation. The runtime per iteration comparison between out-of-core and implicit methods would only be fair if the probability vector is kept in-core, and the matrix alone stored on disk; see [8] for such a comparison.

5 Exploiting Sparsity Structure

In parallel computing, it might be possible to exploit the *sparsity* structure of a matrix while decomposing and distributing the tasks to parallel processors. A few examples of the matrix structures that can be exploited are block-tridiagonal structure, symmetric structures, and *banded* matrices, in which the nonzero elements are confined within a band around the principal diagonal. The Cluster models are an example of banded matrices. The out-of-core solution may also borrow these ideas from parallel computing to obtain better performance. For example, if a large enough Cluster matrix is decomposed into a certain number of square blocks, the Gauss-Seidel calculation of a subset of the probability vector π requires at most two other subsets (blocks) of the vector π. These considerations may be exploited to modify the out-of-core algorithm presented in §4 leading to relatively balanced, regular and lesser amount of disk I/O. Figure 3 explains these facts.

Another sparsity structure that can be exploited by out-of-core and parallel algorithms is present in the matrices for the Polling models. Figure 4 shows the sparsity pattern of the Polling CTMC matrices. We observed that the Polling matrices require just another subset (in addition to the one being updated) of the

[1] `http://www.cs.bham.ac.uk/~dxp/prism/fms.html`

Poll	states	blocks	RAM (MB)	Time (sec/it)	Iter.	MB per π
18	7,077,888	16	13	44	34	54
19	14,942,208	16	27	98	35	114
20	31,457,280	16	62	220	36	240

Table 4. Numerical solution times for the Polling models

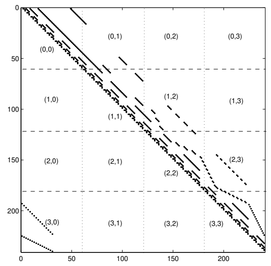

Fig. 4. Sparsity structure of matrices for the Polling models

probability vector π for the Gauss-Seidel computation of a subset of the vector π. We present the results of a modified algorithm which exploits the sparsity structure of the polling models in Table 4. However, the model in Figure 4 retains this structure as long as we keep an upper bound on the number of blocks that the probability vector is decomposed into. For *Poll20* this limit is 16. This is the reason why the number of blocks for all the models in Table 4 are 16; column 3 shows the number of blocks for the vector π. Higher memory requirements for the lower number of blocks can be avoided by dividing the vector π and the matrix Q into a different number of blocks. All matrices in Table 4 are decomposed into 512 or more blocks. The convergence was tested according to equation (3) for $\epsilon = 10^{-7}$. We note in Table 4 that a model with 31 million states can be solved on an average speed workstation (360 MHz) in just over two hours using 62 MB RAM.

6 Future Directions for Research

We have introduced in this paper a new out-of-core algorithm for the numerical solution of large systems of linear equations. The algorithm relaxes the size limitation on implicit and explicit numerical solution methods, caused by the need to store the probability vector(s) explicitly. It can be applied in the numerical solution of any CTMC matrix, for it makes no assumptions about the structure of the matrix. The algorithm employs the compact MSR sparse storage format which itself was introduced in this paper. The scheme offers memory savings over current sparse storage schemes and it can be used with in-core, out-of-core or with parallel iterative numerical solutions of CTMC matrices. However, the correctness of the algorithm does not rely on the memory format employed.

We have described an implementation of the algorithm for an average-sized workstation. The physical memory of the workstation can at most hold one vector of 15 million doubles. By decomposing the matrix and the probability vector into a certain number of blocks, we were able to solve systems with up to 41 million states on a workstation with 128 MB RAM. We have also explained that even larger systems can be solved on a workstation of this average capacity. We further demonstrated how the sparsity structure of various models can be exploited to obtain better performance from the out-of-core algorithm. The paper opens new directions for future research which we outline below.

– The CPU usage for the implementations of the out-of-core algorithms described in this paper is around 60%. The CPU remains idle for 40% of the time due to disk I/O. Symbolic methods such as MTBDDs or MDs provide very compact encodings at a cost of CPU time. We envisage that keeping the matrices in main memory in aforementioned implicit encodings can improve the performance of the out-of-core algorithm. Moreover, this proposed work would also eliminate the hindrance of the implicit methods which is caused by the need to store the probability vector(s) explicitly in the main memory.
– Non-stationary iterative methods such as *Conjugate Gradient Squared* require more than two vectors of size proportional to the number of states in the system. Storing the vector π onto disk and keeping a certain number of vectors of size equal to a subset of the vector π means it is possible to use non-stationary methods even if the memory required to hold complete vectors is not available.
– Distributing a certain iterative computation such as Jacobi to a cluster of workstations means very high communication costs, especially if the link available on the cluster is not fast. Out-of-core methods which store the matrix onto disk and keep the probability vector in main memory have already been parallelised. We believe that the performance of out-of-core methods can be improved by parallelising the out-of-core algorithm introduced in this paper because it will enable very large models to be solved on clusters of modest sizes by reducing the main memory requirement and by providing work to overlap with the idle time during interprocessor communications.
– The out-of-core algorithm presented in this paper suffers from the problems associated with the sparsity structure of the CTMC models, for it causes either of the two Compute and Disk-IO processes to wait while one is busy doing its job. Further research is required to find out better out-of-core algorithms that adapt to the sparsity structure, and are capable of enjoying regular and balanced File I/O.

7 Acknowledgements

We would like to thank Stephen Gilmore for a discussion which motivated us to do this work. Gethin Norman and David Parker are acknowledged for the frequent useful conversations we had with them. We are also grateful to Daniel D. Deavours and William H. Sanders with whom we talked in Aachen.

References

1. Alexander Bell and Boudewijn 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.

2. G. Ciardo 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.
3. G. Ciardo 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.
4. G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.
5. Gianfrance Ciardo. What a Structural World. In Reinhard German and Boudewijn Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 3–16, Aachen, Germany, September 11-14 2001.
6. E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Effificient Data Structure for Matrix Representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, pages 6a:1–15, Tahoe City, May 1993. Also available in *Formal Methods in System Design*, 10(2/3), 1997.
7. Daniel D. Deavours and William 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)*, pages 58–71, St. Malo, France, June 1997. Springer-Verlag.
8. Daniel D. Deavours and William H. Sanders. An Efficient Disk-based Tool for Solving Large Markov Models. *Performance Evaluation*, 33(1):67–84, 1998.
9. Daniel D. Deavours and William H. Sanders. "On-the-fly" Solution Techniques for Stochastic Petri Nets and Extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.
10. S. Donatelli. Superposed Stochastic Automata: A Class of Stochastic Petri Nets Amenable to Parallel Solution. *Performance Evaluation*, 18:21–36, 1993.
11. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications. Clarendon Press Oxford, (with corrections)1997.
12. G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on CAD*, 15(12):1479–1493, December 1996.
13. H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Probability. *Formal Aspects of Computing*, 6:512–535, 1994.
14. B. Haverkort, H. Hermanns, and J.-P. Katoen. On the Use of Model Checking Techniques for Dependability Evaluation. In *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 228–237, Erlangen, Germany, October 2000.
15. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.
16. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. Numerical Solutions of Markov Chains (NSMC'99)*, Zaragoza, 1999.
17. O. Ibe and K. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.
18. William J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1999.
19. William J. Knottenbelt and Peter G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In *Proc. Numerical Solution of Markov Chains (NSMC'99)*, Zaragoza, 1999.
20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. TOOLS 2002*, April 2002.

21. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *Proc. TACAS 2002*, April 2002.
22. Andrew S. Miner. Efficient Solution of GSPNs using Canonical Matrix Diagrams. In Reinhard German and Boudewijn Haverkort, editors, *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 101–110, Aachen, Germany, September 2001.
23. B. Plateau. On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms. In *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 147–153, Austin, TX, USA, May 1985.
24. Yousaf Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.
25. Markus Siegle. Advances in Model Representations. In Luca de Alfaro and Stephen Gilmore, editors, *Proc. PAPM/PROBMIV 2001, Available as Volume 2165 of* LNCS, pages 1–22, Aachen, Germany, September 2001. Springer Verlag.
26. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
27. Sivan Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.