Dual-Processor Parallelisation of Symbolic Probabilistic Model Checking

Marta Kwiatkowska, David Parker, Yi Zhang
({mzk,dxp,yxz}@cs.bham.ac.uk)
School of Computer Science, University of Birmingham,
Edgbaston, Birmingham, B15 2TT, UK

Rashid Mehmood
(Rashid.Mehmood@cl.cam.ac.uk)
Computer Laboratory, University of Cambridge,
Cambridge, CB3 0FD, UK

Abstract

In this paper, we describe the dual-processor parallelisation of a symbolic (BDD-based) implementation of probabilistic model checking. We use multi-terminal BDDs, which allow a compact representation of large, structured Markov chains. We show that they also provide a convenient block decomposition of the Markov chain which we use to implement a parallelised version of the Gauss-Seidel iterative method. We provide experimental results on a range of case studies to illustrate the effectiveness of the technique, observing an average speed-up of 1.8 *with two processors. Furthermore, we present an optimisation for our method based on preconditioning.*

1 Introduction

Probabilistic model checking is an automated technique for the formal verification of systems which exhibit stochastic behaviour. Similarly to conventional non-probabilistic model checking, this is based on the construction of a mathematical model of the system comprising the state space, the set of all possible configurations which the system can be in, and all of the transitions which can occur between these states. In this case, the model also includes information about the probability of each transition occurring at a given time.

In this paper, we consider two types of probabilistic models: discrete-time Markov chains (DTMCs), in which each transition between states is a discrete time-step and is selected according to a discrete probability distribution; and continuous-time Markov chains (CTMCs), where the (real-time) delay before moving from one state to another is modelled by a negative exponential distribution. Properties to be verified for these systems are expressed in temporal logic: PCTL for DTMCs and CSL for CTMCs. This allows specifications such as "the probability of shutdown occurring within 24 hours is at most 0.01" or "in the long-run, the probability that the system is stable is at least 0.75".

As is typically the case with formal verification approaches, one of the main practical problems for probabilistic model checking is the state space explosion problem: the tendency for models of real-life systems to grow prohibitively large, and hence require excessive amounts of memory and/or time to perform. One direction of research which attempts to combat this is *symbolic* approaches, which employ data structures based on binary decision diagrams (BDDs) to generate and manipulate compact representations of extremely large models. In the context of probabilistic model checking, multi-terminal BDDs (MTBDDs) are a commonly used variant. Another direction is the use of *parallel* or *distributed* implementations, which split the problems of storage and computation between several computers or processors. In this paper, we embark on combining the two approaches.

The problem of performing probabilistic model checking for DTMCs and CTMCs comprises a number of tasks: model construction, graph-based algorithms (e.g. determining the set of reachable states of a model), and numerical computation. In our experience, the last of these is typically the bottleneck, and it is on this that we therefore concentrate. Moreover, we focus here on the problem of solving linear equation systems, which allows us to compute both *reachability probabilities* (the probability of reaching a set of states from another state) and *steady-state probabilities* (the long-run behaviour of the system). This allows verification of a wide range of properties of DTMCs and CTMCs.

In this paper, we present techniques to perform the solution of linear equation systems with the Gauss-Seidel iterative method, using MTBDD data structures in a parallel setting. Currently, we are focusing on small-scale parallelisation using shared memory, i.e. we are aiming at desktop machines equipped with dual or quad processors. In future, we plan to extend this work to a more general parallelised setting. We first discuss how using MTBDDs provides us with a convenient decomposition of our models into blocks. Next, we show how this allows an efficient symbolic implementation of Gauss-Seidel, and then how this can be parallelised. Lastly, we present a preconditioning-based optimisation of our approach.

The rest of this paper is organised as follows. In the remainder of this section, we discuss related work. In the subsequent sections, we give a high-level overview of MTBDD-based approaches, describe the implementation of the techniques outlined in the previous paragraph and present experimental results to illustrate their effectiveness.

1.1 Related Work

Numerous approaches to the symbolic implementation of probabilistic model checking and, more generally, the analysis of stochastic models, can be found in the literature; see [19] for a survey of this area. The main symbolic methods applicable to stochastic models are Kronecker methods, matrix diagrams and MTBDD-based techniques. To date, the emphasis has been primarily on sequential implementations. Buchholz et al. presents a parallel block Jacobi algorithm using Kronecker symbolic methods for workstation clusters [5]. Our paper is part of ongoing work to improve the efficiency of these techniques through parallelisation. A complementary direction of research, which will be used to improve our work in future, is the development of out-of-core implementations which use disk-based storage to overcome the limitations of processor memory; see e.g. [13].

In the non-symbolic setting (i.e. using explicit data structures such as sparse matrices), many researchers work on distributed and parallel methods for Markov chains. [6] gives an overview of distributed non-symbolic approaches for analysing CTMCs. In [18], the authors investigated a parallel solution for CTMCs by combining Jacobi and Gauss-Seidel iterative methods. In [4], the authors present a parallel iterative solution for passage time densities in semi-Markov models. Parallel implementations of Jacobi and Conjugate Gradient Squared for solving Markov chains are presented in [12]. Recently, Bell and Haverkort developed distributed out-of-core Jacobi and Conjugate Gradient Squared algorithms for generalised stochastic Petri nets on a cluster of dual-processor workstations [2]. In their paper, they point out that as their parallel solution does not use the second processor on each node efficiently, the overall speedup of their algorithm is limited.

In numerical analysis literatures, there has been a significant amount of work done on parallelisation of iterative numerical solution techniques, including the Gauss-Seidel method (see e.g. [3, 10, 23]). Generally, these approaches work by detecting parts of the computation which can be performed independently and assigning them to different processors. This changes the updating order of Gauss-Seidel, which can hinder the rate of convergence. In a large-scale parallel environment, this cost can be amortised by the speed-up from parallelism, but in the dual- or quad-processor setting which we are aiming at this is difficult to achieve. Our approach does not affect the updating order.

Another direction of research is the use of preconditioning, where the aim is to modify the linear equation system being solved to improve its convergence (see [20] for a recent survey). Traditional preconditioning techniques are not suitable for an MTBDD-based implementation because the modifications would destroy the high-level structure in the matrix and introduce new distinct values, both of which would have a detrimental effect on the compactness of the representation. In this paper, we will introduce a simple preconditioning technique, based on the MTBDD representation and the block decomposition it provides, which improves the convergence rate in many cases.

2 Symbolic Representations and Block Structure

Symbolic approaches to model checking are those which use data structures based on BDDs to arrive at a compact model representation. This is achieved by exploiting structure and regularity, obtained from the high-level description of the model. In practice, symbolic representations of probabilistic models are often orders of magnitude smaller than explicit alternatives such as sparse matrices. In addition to compactness of model representation, other advantages of symbolic approaches include fast and efficient model construction and combination with efficient BDD implementations of non-probabilistic model checking techniques, e.g. computing the set of the reachable states of a model.

A variety of data structures have been proposed for this purpose. Here, we use MTBDDs (multi-terminal binary decision diagrams) [1, 9], which have already proved extremely successful for probabilistic model checking (see e.g. [16]). In the following paragraphs we give a high-level overview of MTBDDs and how we use them, covering only the aspects which are relevant to this paper. For a more in-depth discussion of the data structure and its application to probabilistic model checking, see e.g. [15, 17, 22].

An MTBDD is a reduced binary tree which represents a real-valued function over Boolean variables. The transition probability matrix of a DTMC or the transition rate matrix of a CTMC is essentially a function mapping pairs of states to real numbers. Hence, by choosing an encoding of the set of states into n Boolean variables, we can store this information as an MTBDD over $2n$ variables. The nodes of the MTBDD are grouped into horizontal levels. The top $2n$ levels are labelled with Boolean variables, one level for each variable. The bottom level is labelled with real values. To determine the probability or rate associated with a pair of states, we trace a path from the top to the bottom of the tree, resolving the binary decision at each node according to the value of the corresponding Boolean variable in the encodings of the state pair. The value at the bottom of the tree gives the result.

MTBDDs are a recursive data structure. While the top (root) node of the tree represents the whole matrix being stored, by descending two levels (corresponding to one pair of Boolean variables), we reach nodes representing the 4 quadrants of the matrix. More generally, descending $2i$ levels splits it into $(2^i)^2$ submatrices. Essentially, each node of the MTBDD represents a submatrix of the overall matrix; the lower the node in the tree, the smaller the submatrix. For a given value of i, the nodes on this level provide a very convenient partitioning of the whole matrix into blocks.

We observe that the compactness of the MTBDD data structure derives from the fact that the tree is stored in reduced form, with identical subtrees being merged. This corresponds to identical submatrices being merged. Through the use of heuristics, it is possible to derive an extremely efficient encoding of the model, whereby many submatrices are repeated and hence only stored once.

In fact, the matrix partitioning described above is somewhat more complicated. In practice, we are only interested in the *reachable* states of the probabilistic model, i.e. those from which there is a path from the initial state. The unreachable states, which often significantly outnumber the reachable ones, can and, for efficiency reasons, should be ignored. We set the corresponding rows and columns to zero. Since the distribution of unreachable states is unpredictable, the decomposition into submatrices in fact produces blocks of uneven size. Furthermore, transition matrices are typically very sparse, with many entries equal to zero, and hence many of the submatrices will be empty (all zero). Note that it is essential to keep the unreachable states in the overall MTBDD encoding; removing them would destroy the regularity of the data structure and hence its compactness.

While MTBDDs provide a compact representation, access to the matrix elements, particularly if required row-by-row or column-by-column is much slower than an explicit storage scheme such as a sparse matrix. In this paper, we adopt a compromise between the two approaches. We generate our matrix as an MTBDD, select a level of the tree, and generate an explicit (sparse matrix) representation of the corresponding matrix block for each node on that level, essentially partitioning the whole matrix into a number of blocks. Note that this is not equivalent to storing the whole matrix explicitly because many matrix blocks are repeated and we only store them once each. In addition, we store explicit information about where the matrix blocks occur in the overall matrix. In fact, this is just another matrix, storing pointers to submatrices rather than the actual values.

Note that, although the resulting data structure is really two layers of entirely explicit storage, we still refer to this as a symbolic storage scheme since it relies on the structure and regularity of the MTBDD from which it was derived to remain a compact representation. Later in the paper, we will present experimental results to illustrate that it is far more compact than a single sparse matrix.

This data structure will be used in the remainder of the paper. From this point on, however, the manner in which it was created is not important: we need only know that we have a compact representation of a large matrix which is divided into blocks and that we have efficient (row-wise) access both to the blocks themselves and the elements within them. In the next section, we describe how this allows us to parallelise a symbolic implementation of Gauss-Seidel.

3 Parallelising Gauss-Seidel

3.1 The Gauss-Seidel Method

We first describe a technique for implementing Gauss-Seidel. In fact, this method is applicable to any storage scheme but is designed to suit the scheme outlined in the previous section. We consider the problem of solving a linear equation system $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{b}$ is vector of length n. In the context of probabilistic model checking, $\mathbf{A}$ will have been derived from the transition matrix representing the probabilistic model. We denote the (i, j)th element of matrix $\mathbf{A}$ as $\mathbf{A}_{ij}$ and the ith element of vector $\mathbf{b}$ as $\mathbf{b}_i$.

Gauss-Seidel works by, at each iteration, computing a new approximation to the solution vector $\mathbf{x}$, using the matrix $\mathbf{A}$, the vector $\mathbf{b}$ and the values of $\mathbf{x}$ from the previous iteration. The algorithm in Figure 1 performs the numerical computation for one iteration of Gauss-Seidel. Note that we require only one copy of the solution vector $\mathbf{x}$, which we simply overwrite as we compute its new entries.

Assume now that we have a division of the matrix $\mathbf{A}$ into $N \times N$ blocks. We denote the (p, q)th block of $\mathbf{A}$ as $\mathbf{A}_{(pq)}$ and, correspondingly, the pth block of vector $\mathbf{b}$ as $\mathbf{b}_{(p)}$. The (i, j)th element of submatrix $\mathbf{A}_{(pq)}$ is $\mathbf{A}_{(pq)ij}$ and the ith element of subvector $\mathbf{b}_{(p)}$ is $\mathbf{b}_{(p)i}$. The size of block $\mathbf{A}_{(pq)}$ is defined as $n_p \times n_q$, and hence the size of $\mathbf{b}_{(p)}$ is n_p. We

```
1. for $(0 \le i < n)$
2.     $\mathbf{x}_i := (\mathbf{b}_i - \sum_{0 \le j < n, j \ne i} \mathbf{A}_{ij} \cdot \mathbf{x}_j)/\mathbf{A}_{ii}$
```

Figure 1. A simple algorithm to perform an iteration of Gauss-Seidel.

```
1. for $(0 \le p < N)$
2.     $\mathbf{temp} := \mathbf{b}_{(p)}$
3.     for each block $\mathbf{A}_{(pq)}$ with $q \ne p$
4.         $\mathbf{temp} := \mathbf{temp} - \mathbf{A}_{(pq)}\mathbf{x}_{(q)}$
5.     for $(0 \le i < n_p, i \ne j)$
6.         $\mathbf{x}_{(p)i} := (\mathbf{temp}_i - \sum_{0 \le j < n_p} \mathbf{A}_{(pp)ij} \cdot \mathbf{x}_{(p)j})/\mathbf{A}_{(pp)ii}$
```

Figure 2. A block-based reformulation of the algorithm for an iteration of Gauss-Seidel.

can now formulate the computation for a single iteration of Gauss-Seidel as shown in Figure 2.

The pth phase of the outer loop computes updates to the values of block $\mathbf{x}_{(p)}$ of the solution vector. Note that we still only store one copy of $\mathbf{x}$ but we use an additional vector $\mathbf{temp}$ of size $\max_{0 \le p < N}\{n_p\}$ to accumulate the multiplications of submatrices and subvectors. Observe also that the iteration over matrix blocks in line 3 need only be over non-empty blocks.

The appeal of this reformulation is that it requires isolated access to individual matrix blocks. The pth phase of an iteration (lines 2–6) uses only the pth row of blocks of $\mathbf{A}$, of which lines 3–4 iterate over the non-diagonal blocks one by one, and lines 5–6 use the diagonal block $\mathbf{A}_{(pp)}$. In similar fashion, vector blocks are also accessed one at a time.

It should now be clear how the algorithm above, while potentially also suitable for any matrix representation, is particularly appropriate for the symbolic storage scheme outlined in the previous section, which provides fast (row-wise) access both to each matrix block and to each element of that block. The algorithm will now prove amenable to parallelisation, as we will show in the next section. Furthermore, we note that this is the first time that an efficient Gauss-Seidel algorithm has been presented for an MTBDD-based matrix representation since, in its conventional form, it does not allow efficient access to individual rows.

3.2 Parallelisation

Our block-based reformulation of the Gauss-Seidel algorithm also facilitates its parallelisation. Assume that we have P processes, with IDs $0, ..., P-1$. The basic idea is to distribute the outermost loop (over p) of Figure 2 among the P processes. Process k will compute the new values for the subvectors $\{\mathbf{x}_{(k)}, \mathbf{x}_{(k+P)}, \ldots\}$. The parallel algorithm for process k is described in Figure 3.

During every round, each process k computes some subvector $\mathbf{x}_{(p)}$ using the pth row of matrix blocks, i.e. $\mathbf{A}_{(pq)}$ for $q = 0, \ldots, N-1$. For each matrix block, it requires the corresponding vector block $\mathbf{x}_{(q)}$ to perform a matrix-vector multiplication. To ensure that the block-based algorithm remains faithful to the original Gauss-Seidel algorithm, we have to take special care with vector blocks which are being computed by other processes in the same round. More specifically, we need up-to-date values of the vector-blocks for the cases where $q \le p$. For $q > p$, this is not the case. Furthermore, for $q < p - k$, the vector blocks have already been updated in previous rounds. Hence, we can safely start by dealing with these matrix blocks. For the remaining blocks, i.e. those in the range $p - k \le q < p$, we need to wait for the process which is computing that block (i.e. process $p - q$) to finish computing it before we can use it. This procedure can be observed by the replacement of lines 3–4 from Figure 2 with lines 3-7 in Figure 3. Finally, at the end of the round, process k has to wait for the other processes to finish before processing to the next round (see line 10 of Figure 3). Note also that although the algorithm effectively cycles through all N blocks in each row of blocks in the matrix, in practice, many of these blocks will be empty (all zero) and can be ignored.

3.3 Preconditioning

Next, we introduce an improvement to the algorithms described above based on preconditioning. This method works by modifying the order in which the subvectors of the solution vector are updated in each iteration. It is inspired by traditional preconditioning methods for Gauss-Seidel, most of which perform a kind of incomplete LU

```
1.   for ($p = \{k, k+P, \ldots\}$)
2.       $\mathbf{temp} := \mathbf{b}_{(p)}$
3.       for each block $\mathbf{A}_{(pq)}$ with $q < p - k$ or $q > p$
4.           $\mathbf{temp} := \mathbf{temp} - \mathbf{A}_{(pq)}\mathbf{x}_{(q)}$
5.       for each block $\mathbf{A}_{(pq)}$ with $p - k \le q < p$
6.           wait for process $p - q$
7.           $\mathbf{temp} := \mathbf{temp} - \mathbf{A}_{(pq)}\mathbf{x}_{(q)}$
8.       for ($0 \le i < n_p, i \ne j$)
9.           $\mathbf{x}_{(p)i} := (\mathbf{temp}_i - \sum_{0 \le j < n_p} \mathbf{A}_{(pp)ij} \cdot \mathbf{x}_{(p)j}) / \mathbf{A}_{(pp)ii}$
10.      synchronise with other processes
```

Figure 3. Parallel algorithm for an iteration of Gauss-Seidel (process ID=k).

factorisation. A full LU decomposition would reduce the number of iterations to one; an incomplete decomposition can still provide a useful improvement in the convergence. In the resulting U matrix of a full LU decomposition, the number of non-zero elements in each row tends to decrease from the top to the bottom of the matrix. In our preconditioning method, we try to achieve the same result, but at the level of matrix blocks.

We first define a Boolean $N \times N$ matrix $\mathbf{B}$, representing the MTBDD block structure, i.e. $\mathbf{B}_{pq}$ is 1 if matrix block $\mathbf{A}_{(pq)}$ is non-empty and 0 otherwise. We then define $B_p = \sum_{0 \le q < N} \mathbf{B}_{pq}$ for any $0 \le p < N$. In the block-based Gauss-Seidel algorithm of the previous section, each iteration updates the N subvectors of the solution vector $\mathbf{x}$ in the order $0, 1, \ldots, N-1$. To apply preconditioning, we instead update them in the order $i_1, i_2, ..., i_N$, where $B_{i_1} \ge B_{i_2} \ge ... \ge B_{i_N}$. In the next section, we will show how this approach can reduce the number of iterations required for convergence.

4 Experimental Results

We now present experimental results to evaluate the performance of the parallelised Gauss-Seidel method from the previous sections. We implemented our methods by integrating them into the PRISM 2.0 model checker [14]. We used a selection of case studies, including three CTMC models (a flexible manufacturing system (FMS) [8], a Kanban manufacturing system [7] and a workstation cluster [11]) and a DTMC model (multiplexing of unreliable NAND gates) [21].

For the CTMCs, we model checked CSL properties which required computation of the steady-state probabilities; for the DTMC, we verified a PCTL property which required computation of reachability probabilities. For each example, we can generate models of varying size by changing one or more parameters. For more information about the examples, their parameters and the corresponding properties, see the PRISM web site (www.cs.bham.ac.uk/~dxp/prism).

First, we give some statistics to show the memory usage of the MTBDD-based storage scheme we use. As discussed earlier in the paper, by selecting different levels of the MTBDD, we can create different decompositions of the matrix into blocks, which will have varying memory consumption. Figure 4 shows the variation in memory usage for a selection of models. As expected, for levels either very low or very high in the MTBDD, we get a much higher memory usage. In the former case, we are having to store very large submatrices explicitly; in the latter, we are creating very small submatrices, but the information about the structure of the blocks is itself a very large matrix. In fact, for each graph, the left-most point plotted is equivalent to the creation of a fully explicit (sparse matrix) representation.

The magnitude of the drop in the middle section emphasises how compact our representation remains, compared to the explicit version. It can also be seen that there is a wide range of values in the middle of the graph for which we can achieve such a reduction. For the results in the remainder of this section, we manually selected a suitable level. In future, we plan to investigate this area further and derive appropriate heuristics.

Next, we show timing statistics for model checking using four separate implementations: sequential symbolic versions of the Jacobi and Gauss-Seidel methods, the first of which was already available in PRISM 2.0, and the second of which is now available with our new formulation; and the parallelised version of Gauss-Seidel, with and without preconditioning.

Our experiments were performed on an SMP PC with two 2.0 GHz Intel Pentium 4 Xeon processors and 1 GB of main memory running Linux. Parallelisation is implemented with threads, using the pthread package. Storage in memory of the matrix $\mathbf{A}$ and vector $\mathbf{b}$ describing the linear equation system is shared between threads but since only

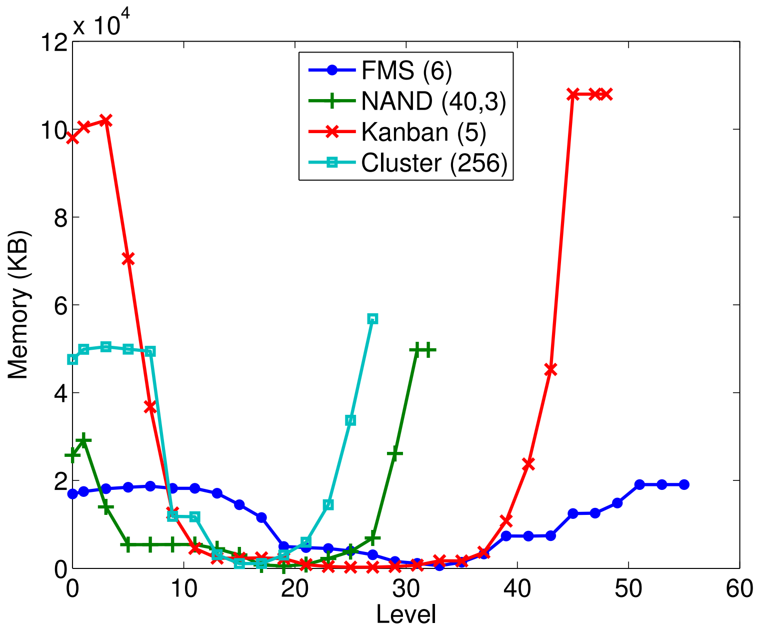

Figure 4. Memory for matrix storage using different block decompositions.

Table 1. Total execution times (seconds) for each implementation.

Model	States	Sequential Jacobi	Sequential GS	Parallel GS	Parallel GS (pre.)
FMS (N=6)	537,768	128.3	110.6	55.1	49.7
FMS (N=7)	1,639,440	461.0	413.9	220.5	191.5
FMS (N=8)	4,459,455	1800.7	1294.7	724.3	620.4
Kanban (N=5)	2,546,432	346.5	268.8	149.0	127.1
Kanban (N=6)	11,261,376	2087.8	1688.0	945.7	828.6
Cluster (N=256)	2,373,652	1144.6	334.3	257.2	257.7
NAND (N=40,M=3)	1,004,821	103.6	178.2	88.8	88.9
NAND (N=40,M=5)	2,003,041	408.9	590.8	285.6	284.8
NAND (N=40,M=7)	3,001,261	913.5	1181.4	590.0	589.7
NAND (N=40,M=9)	3,999,481	1618.0	1978.7	999.9	998.7

read-access is required, no synchronisation is performed. In fact, the same is true of the solution vector $\mathbf{x}$ since, although it is both read and written by all threads, different parts of the vector are accessed by different processors all the time during the execution. The synchronisation referred to in lines 6 and 10 of Figure 3 is implemented with the condition variable mechanism provided in the pthread package.

Table 1 shows the actual (total) run-times for our implementations. Figure 5 shows the speed-up of each approach, relative to the sequential Gauss-Seidel case. Our first observation is that our new (sequential) implementation of Gauss-Seidel is generally faster than the old Jacobi implementation. This is to be expected since it usually requires less iterations to converge (in fact, for the NAND example, the convergence rate is unchanged and the overhead in computation means Gauss-Seidel is actually slightly slower).

Secondly, we note that our parallelisation of the Gauss-Seidel algorithm is effective: the average speed-up over these ten benchmark examples is approximately 1.8. Finally, we see that in half of the cases the preconditioning provides a further improvement. This is due to the reduction in iterations for convergence required. For the NAND and workstation cluster examples, the convergence rate is

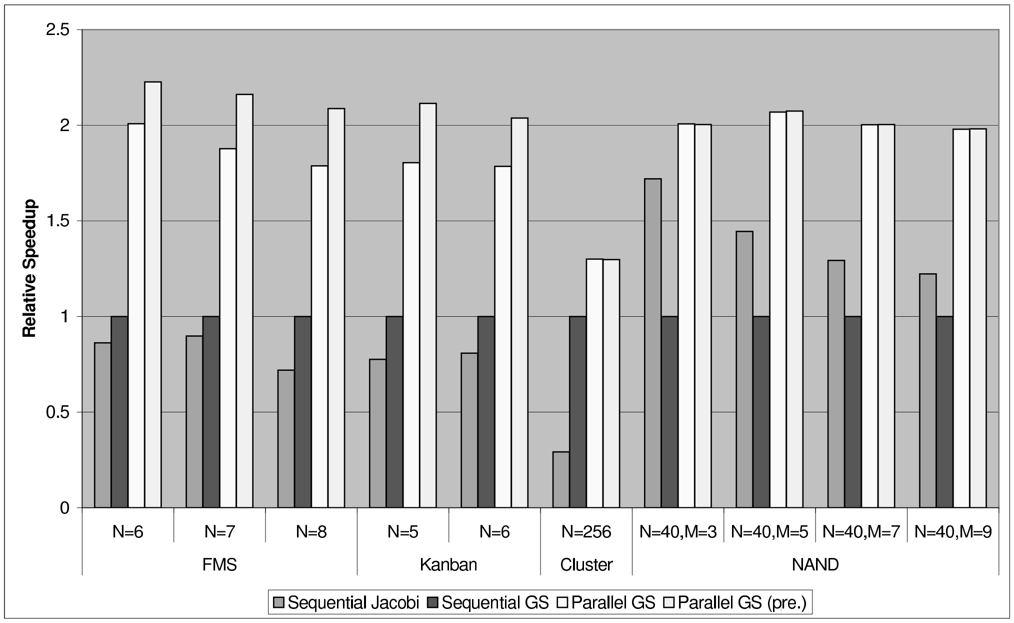

Figure 5. Speed-up in execution time for each implementation.

Table 2. Changes in convergence rate due to preconditioning.

Model	Iterations	
	Original	Precond.
FMS (N=6)	812	750
FMS (N=7)	966	879
FMS (N=8)	1,125	1,059
Kanban (N=5)	461	460
Kanban (N=6)	622	594
Cluster (N=256)	1002	1002
NAND (N=40,M=3)	480	480
NAND (N=40,M=5)	800	800
NAND (N=40,M=7)	1120	1120
NAND (N=40,M=9)	1440	1440

unchanged. This is because each row of blocks in the matrix **A** has the same number of non-zero blocks and hence our preconditioning technique has no effect. We give the precise figures for the convergence of each example in Table 2.

5 Conclusions and Future Work

In this paper, we have presented an MTBDD-based method for the Gauss-Seidel algorithm and its parallelisation, with a target platform of shared memory dual- or quad-processor workstations. We implemented our technique in the probabilistic model checking tool PRISM and, on a range of examples, illustrated an average speed-up of 1.8 using two processors. We were also able to optimise our approach further with a simple notion of preconditioning.

Presently, grid systems are attracting a lot of attention. Nodes of grid systems are usually clusters of PCs with dual-processors, connected by a high-speed network. From an architecture point of view, such systems exhibit properties of both shared memory systems and message passing systems. In future, we are planning to fully parallelise our approach for use on grid systems, by investigating a combination of the dual-processor approach in this paper with other parallel technologies for the message passing model.

We would like also to investigate MTBDD-based preconditioning methods. It is widely accepted that preconditioning methods are essential for the efficiency of iterative algorithms. MTBDD-based implementations of iterative solution methods require preconditioning methods which preserve the regularity of the data structure used for matrix storage. We have demonstrated one such approach in this paper. We plan to continue developing efficient techniques to support preconditioning for symbolic implementations.

Acknowledgements

The work in this paper was supported in part by EPSRC grants GR/S11107 and GR/S27252.

References

[1] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conference on Computer-Aided Design (ICCAD'93)*, pages 188–191, 1993.

[2] A. Bell and B. R. Haverkort. Serial and parallel out-of-core solution of linear systems arising from generalised stochastic petri nets. In *Proceedings of High-Performance Computing Symposium (HPC), Advanced Simulation Technologies Conference (ASTC 2001), Seattle, USA*, pages 242–247. Society for Computer Simulation, Apr. 2001.

[3] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[4] J. Bradley, N. Dingle, W. Knottenbelt, and H. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-markov models. In *4th International Meeting on the Numerical Solution of Markov Chains (NSMC 2003)*, pages 99–120. Chicago, USA, Sept. 2003.

[5] P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using kronecker algebra. In *Numerical Solutions of Markov Chains (NSMC'99)*, pages 76–95. Prensas Universitarias de Zaragoza, Sept. 1999.

[6] G. Ciardo. Distributed and structured analysis approaches to study large and complex systems. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 344–374. Springer-Verlag, Nov. 2001.

[7] 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.

[8] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[9] E. Clarke, M. Fujita, P. McGeer, K. McMillan, 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)*, pages 1–15, 1993.

[10] G. H. Golub, J. M. Ortega, and G. Golub. *Scientific Computing: An Introduction With Parallel Computing*. Academic Press, 1993.

[11] 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.

[12] W. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, University of London, Imperial College of Science, 1999.

[13] M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A symbolic out-of-core solution method for Markov models. In *Proc. Workshop on Parallel and Distributed Model Checking (PDMC'02)*, volume 68.4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[15] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[16] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. Springer, 2001.

[17] R. Mehmood, D. Parker, and M. Kwiatkowska. An efficient BDD-based implementation of Gauss-Seidel for CTMC analysis. Technical Report CSR-03-13, School of Computer Science, University of Birmingham, December 2003.

[18] V. Migalln, J. Penads, and D. B. Szyld. Experimental study of parallel iterative solutions of markov chains with block partitions. In *Numerical Solutions of Markov Chains (NSMC'99)*, pages 96–110. Prensas Universitarias de Zaragoza, Sept. 1999.

[19] A. Miner and D. Parker. Symbolic representations and analysis of large probabilistic systems. In C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, M. Siegle, and F. Vaandrager, editors, *Lecture Notes in Computer Science Tutorial Volume: Validation of Stochastic Systems*. Springer, 2004. To appear.

[20] H. Niki, K. Harada, M. Morimoto, and M. Sakakihara. The survey of preconditioners used for accelerating the rate of convergence in the Gauss-Seidel method. *Journal of Computational and Applied Mathematics*, 164(1):587–600, Mar. 2004.

[21] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. Evaluating the reliability of defect-tolerant architectures for nanotechnology with probabilistic model checking. In *Proc. International Conference on VLSI Design (VSLI'04)*, pages 907–914. IEEE Computer Society Press, 2004. To appear.

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

[23] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, 1996.