DISK-BASED TECHNIQUES FOR EFFICIENT SOLUTION OF LARGE MARKOV CHAINS

by

RASHID MEHMOOD

A thesis submitted to the Faculty of Science of
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
Faculty of Science
University of Birmingham
October 2004

Abstract

Computer systems are ubiquitous in almost all spheres of our life, motivating the need for them to function correctly and in a timely manner. Continuous time Markov chains (CTMCs) are a widely used formalism for the performance analysis of computer systems. A large variety of useful performance measures can be derived from a CTMC via the computation of its steady-state probabilities. Traditional methods for performance analysis typically require the generation and storage of the underlying state space of the CTMC, and the processing of the state space for the numerical solution. CTMC models for even trivial real-life systems are usually huge, and both the amount of required memory and the time to compute the solution pose a major difficulty.

In this thesis, we present techniques which extend the size of analysable models on a single workstation, with the goal for alleviating the state space explosion problem. We introduce explicit and symbolic disk-based methods which relax the storage limitations of contemporary techniques and are able to extend the size of solvable models by an order of magnitude. We also propose a new sparse storage scheme which provides 30% or more savings over the conventional sparse schemes and improves the solution speed. Furthermore, we present modifications to multi-terminal binary decision diagrams (MTBDDs), a symbolic data structure for storing CTMCs. The modifications improve the time and memory properties of this data structure, and allow an efficient implementation of the Gauss-Seidel iterative method, which was not possible previously.

Using the techniques introduced in this thesis, we demonstrate analysis of models with over 1.2 billion states and 16 billion transitions on a single workstation. Currently, models of such sizes cannot be solved on a contemporary workstation using conventional techniques.

Acknowledgements

I would like to thank my supervisor, Marta Kwiatkowska, for her motivation and continuous, tireless advice. Marta Kwiatkowska, Gethin Norman and David Parker are all acknowledged for many discussions, for proof reading this thesis and other manuscripts, and for many other things – I am very grateful to them for their help, support and suggestions.

Achim Jung, Georgious Theodoropoulos and Tom Axford are my thesis group members. They are acknowledged for their time, effort and guidance during this PhD work. I have learnt many things from Markus Siegle and Alexander Bell during the write up of a survey paper. I am very thankful for the many fruitful discussions I had with them. Markus Siegle and Mark Ryan are also acknowledged for their valuable comments on this thesis. Thanks must also go to the School of Computer Science for funding my PhD studies. I would also like to thank Stephen Gilmore for a discussion which helped me to direct myself for this work.

Many people have contributed in my life, directly or indirectly, helping me to reach this stage. First and foremost are my parents and my teachers whose efforts cannot be easily acknowledged. Another name which I would like to mention here is of my brother Raheel Shah. I am also thankful to my wife for her patience during the write up of this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

Introduction

Computer and communication systems are being used in a variety of applications ranging from financial institutions, information services to aeroplane, railway and car control systems. The complexity of these systems is rising due to the amount of concurrency and communication among their components. The applications of these systems to safety and security critical entities, in particular, motivates the need for these systems to function safely, correctly and in a timely manner.

Discrete-state models have proved to be a valuable tool in the performance analysis of 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 (the *state space*), and identifying the transitions which may occur among the states of the system. Since many real-life systems, or the environments in which they operate, are inherently stochastic, it is desirable or even necessary to include these probabilistic aspects in their models. Often, a system under study can be modelled as a continuous time Markov chain (CTMC). CTMCs are widely used in modelling and performance analysis. Typically, the models are represented as matrices consisting of rates, the parameters of exponential distributions. Performance characteristics such as mean waiting time or throughput are obtained from so called steady-state probabilities, i.e. the probability of being in each state in the long run. This thesis focuses on the calculation of the steady-state probability vector, which can be obtained from the solution of a system of a

linear equations. The size of this system equals the number of states.

A major hurdle associated with the CTMC-based performance analysis is the so called *state space explosion* or the *largeness* problem. This is caused by the fact that a system is usually composed of a number of concurrent subsystems, and that the size of the state space of the overall system is generally exponential in the number of subsystems. This means that the number of states for even a trivial real-life system is substantial. Consequently, much research is focused on the development of techniques, i.e. methods and data structures, which minimise the computational (space and time) requirements for analysing large and complex systems.

There are a number of techniques that attempt to reduce the amount of storage required for CTMCs, or avoid the storage altogether. Here, we assume that the CTMC matrix has been generated and will be used for the numerical solution of the steady-state vector. The conventional approach is to use the *explicit* storage of the state space and associated data structures, employing *sparse* storage techniques inherited from the linear algebra community. Standard numerical algorithms can thus be used for the solution. Another direction in this context comprises those techniques which rely on exploiting the regularity and structure in models, and hence provide an *implicit*, usually compact, representation for large models. In the worst case, however, such approaches may result in a larger storage requirement than the explicit methods. A subcategory among the implicit methods is *symbolic methods*, i.e. those usually based on binary decision diagrams (BDDs) and extensions thereof. These implicit techniques include special routines to manipulate the implicit data structures during the process of performance analysis. Essentially, the difference between the implicit and explicit approaches lies in that the former relies on generating a compact storage of the CTMC matrix by exploiting structure and regularity in the CTMC, while the latter stores and manipulates the entire matrix explicitly. Both implicit and explicit techniques have been strengthened by *parallel* and *distributed* techniques, which use resources of shared or distributed memory computers to store and analyse a model.

Our aim in this dissertation is to combat the state space explosion problem for CTMC analysis. We develop techniques which can extend the size of models analysable on a single workstation. To realise this, we have employed

both implicit and explicit representations. In particular, we focus on *out-of-core* techniques which use disk to store all or part of the data structures. The so called *disk-based* or out-of-core techniques have been in use for a long time, but this is the first time they have been used in combination with symbolic data structures. These are designed to achieve high performance when their data structures are stored on disk.

We propose to augment the implicit and explicit approaches with two new methods: first, the *complete out-of-core* method based on a sparse disk-based storage of all data structures; and, second, the *symbolic out-of-core* method, which combines implicit and disk-based approaches. The symbolic data structure we use for the second proposed approach is a BDD variant known as offset-labelled MTBDDs. We also propose a new sparse storage scheme, called the *compact MSR* format, which provides significant memory savings and the solution based on this scheme is typically faster than for the existing sparse schemes. Moreover, we propose modifications to the offset-labelled MTBDD data structure, improving its time and space requirements as well as addressing one of its major limitations regarding the use of iterative solution methods, namely the lack of an efficient Gauss-Seidel method.

We support our proposed techniques with an extensive analysis of their implementations, and comparisons of these techniques with conventional approaches. The benchmark case studies used in our work have been generated using the tool PRISM which is briefly described in Appendix A. The case studies are briefly described in Appendix B. The work presented in this thesis is part of an ongoing effort to improve the range of solution techniques supported by PRISM, and the software produced will shortly be integrated within the tool itself. The source code for the out-of-core methods introduced in this thesis can be found on the Web page http://www.cs.bham.ac.uk/~rxm/.

We use a workstation of modest specifications (440 MHz CPU, 512MB RAM, 6GB disk) for our implementations and experiments. We solve models with up to 384 millions states and 4 billion transitions on this workstation. Furthermore, using a relatively powerful workstation (2.8 GHz CPU, 1GB RAM, 60GB disk), we demonstrate the feasibility of an analysis of a model with over 1.2 billion states and 16 billion transitions. Currently, models of such sizes cannot be solved on a contemporary workstation using conventional techniques.

1.1 Thesis Layout

This dissertation is organised as follows:

Chapter 2 introduces the background material related to this dissertation. The numerical solution methods used for the analysis are discussed. 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 I/O determines the overall solution time for out-of-core methods. The main storage schemes for sparse matrices are reviewed. The offset-labelled MTBDD data structure, which is used in our work presented later in this dissertation, has also been studied in this chapter. The latter part of Chapter 2 surveys related work. It outlines the research which has already been done in this area and how the work in this dissertation contributes to it.

Chapter 3, along with next two chapters, presents the main contributions of this dissertation. This chapter introduces the compact MSR storage scheme and two explicit out-of-core methods. The explicit methods explained in this chapter are analysed and compared with the help of experimental results from their implementations.

Chapter 4 introduces the symbolic out-of-core method. Two implementations of this method are described and compared using the collected results. Furthermore, the method is compared with the explicit methods detailed in Chapter 3 using their implementations on an identical workstation.

Chapter 5 introduces modifications to offset-labelled MTBDDs. Both in-core and out-of-core implementations of the MTBDD-based solution are realised. The two implementations are analysed in detail, and are compared with the explicit and symbolic methods presented in the earlier part of the dissertation.

Finally, in Chapter 6, we summarise and evaluate our work, outline ideas for future research, and conclude.

1.2 Publications

Some of the work presented in this dissertation has already appeared in joint-authored publications. The complete out-of-core method and the compact MSR scheme described in Chapter 3 were introduced in [KM02]. Two implementations of the symbolic out-of-core method described in Chapter 4 were presented in [KMNP02] and [MPK03a]. The improvement to the offset-labelled MTBDD data structure, as described in Chapter 5, appeared as [MPK03b]. The main ideas for the compact MSR, out-of-core algorithms and the two-layer data structure introduced in these four papers are due to the author, as are their implementation and analysis. Marta Kwiatkowska, Gethin Norman and David Parker have all provided guidance throughout. David Parker implemented the original version of the offset-labelled MTBDD data structure that was extended as a result of this thesis. He has also incorporated the resulting source code into the tool PRISM. Gethin Norman developed the case studies used as benchmarks. A recent paper [KPZM04] reports on shared memory dual-processor parallelisation of the Gauss-Seidel method using the offset-labelled MTBDD data structure. The parallel implementation is not the work of the author, but the data structure used is based on the contribution of this thesis.

A survey paper was also written for inclusion in the handbook entitled "Validation of Stochastic Systems", which appeared as [Meh04].

CHAPTER

TWO

Background Material and Related Work

In this chapter, we introduce the background material and the work from the literature which is related to the topic of this thesis. We first explain the overall process of Markov modelling in Section 2.1. In this section, we also explain CTMCs (Section 2.1.1) and define the numerical problem (Section 2.1.2) which we have focused on here. In this thesis, we consider iterative solution methods for the numerical solutions of a system of linear equations, typically the steady-state equations. These are covered in four sections. We begin with an introduction to basic iterative methods in Section 2.2, and move on to review block iterative methods and Krylov subspace methods in Section 2.3 and 2.4, respectively. In Section 2.5, we discuss few widely used convergence criteria for iterative methods.

Analysis of CTMC models usually involves the generation and storage of their state space. We concentrate on two representations for CTMCs, explicit and implicit. The explicit representations are based on sparse storage scheme; we review notable sparse schemes in Section 2.6. The implicit storage we have considered, called offset-labelled MTBDDs, is described in Section 2.7. Finally, we review the work related to the subject of this thesis in Section 2.8. The related work is discussed in two sections. In Section 2.8.1, we review the work carried out mainly for a single (CPU) workstation. In Section 2.8.2, we summarise the approaches targeted for multiple CPUs or workstations.

2.1 Markov Modelling

Computer and communication systems are ubiquitous in all spheres of our life. 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. In this thesis we concentrate on continuous time Markov chains.

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 the system under study. 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 state space needs to be generated from this specification. This excludes product form queueing networks [BCMP75]; *on-the-fly* techniques [DS98b], to some extent, are also an exception. In this thesis, we assume that a CTMC has already been generated, and therefore do not discuss state space generation algorithms and techniques. For details on state space generation algorithms and techniques, see e.g. [CCM95], [ADK97], [CGN98], [KMHK98],

[HBB99], [GMS01], and [Cia01a]. We define continuous time Markov chains in Section 2.1.1. In Section 2.1.2, we formally define the numerical problem which we have considered in this thesis.

2.1.1 Continuous Time Markov Chains

As mentioned in the previous section, a system may be represented as a *stochastic process* by describing the set of different states the system may occupy and by identifying the transitions which can occur between the various states of the system. A stochastic process is a family of random variables $\{X(t), t \in T\}$ indexed by t, usually a time parameter. The random variable $X(t)$ denotes an observation of the system at time instant t. A stochastic process with discrete time indices, for example, $T = \{0, 1, 2, \cdots\}$, is called a *discrete time parameter* stochastic process; if time is continuous, e.g. $T = \{0 \leq t \leq +\infty\}$, it is a *continuous time parameter* stochastic process. The values that random variable $X(t)$ can take are called *states*, and the set of all possible values constitutes the *state space* of the process. If the values assumed by the variable $X(t)$ are discrete, it forms a *discrete* state space.

A *Markov process* is a stochastic process which satisfies the *Markov property*, i.e. for all positive integers k, any sequence of time instances $t_0 < t_1 < \cdots < t_k$ and states $x_0, \cdots, x_k$:

$$P[X(t_k) \leq x_k | X(t_{k-1}) = x_{k-1}, \cdots, X(t_0) = x_0] = P[X(t_k) \leq x_k | X(t_{k-1}) = x_{k-1}]. \tag{2.1}$$

The Markov property formulated above, sometimes known as the *memoryless property*, implies that the state in which the system finds itself at time t_k depends only on the state of the system at time t_{k-1}, while the state occupied by the system at any previous time instances (i.e. $t_0, t_1, \cdots, t_{k-2}$) is completely irrelevant. Consequently, the future state of the system is also independent of the time spent so far in the current state of the system. Note that it is still possible for the transitions to depend on the actual time at which they occur. In this thesis, however, we only consider the *homogeneous* case where the transitions are independent of time.

A Markov process with a discrete state space is referred to as a *Markov chain*. Accordingly, a Markov chain with a continuous time parameter (t) is called a *continuous time Markov chain* (CTMC), and one with a discrete

time parameter is called a *discrete time Markov chain* (DTMC). We focus on CTMCs.

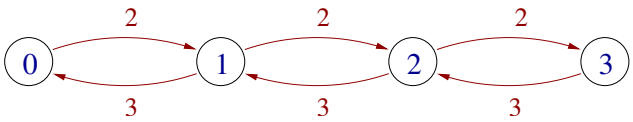
A CTMC is a continuous time, discrete-state stochastic process, i.e. it is a Markov chain which can change state at any time instant. Mathematically, a CTMC is a stochastic process $\{X(t), t \geq 0\}$ which satisfies the Markov property given by Equation (2.1), which in this case (continuous state) can be formulated as:

$$P[X(t_k) = x_k | X(t_{k-1}) = x_{k-1}, \cdots, X(t_0) = x_0] = P[X(t_k) = x_k | X(t_{k-1}) = x_{k-1}], \tag{2.2}$$

for all positive integers k, any sequence of time instances $t_0 < t_1 < \cdots < t_k$ and states $x_0, \cdots, x_k$. The only continuous probability distribution which satisfies the Markov property is the exponential distribution.

A CTMC may be represented by a set of states S, and the *transition rate matrix* $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$. A transition from state i to state j is only possible if the matrix entry $r_{ij} > 0$. The matrix coefficients determine transition probabilities and state *sojourn times* (or *holding times*). Given the *exit rate* of state i, $E(i) = \sum_{j \in S,\, j \neq i} r_{ij}$, the mean sojourn time for state i is $1/E(i)$, and the probability of making transition out of state i within t time units is $1 - e^{-E(i) \cdot t}$. When a transition does occur from state i, the probability that it goes to state j is $r_{ij}/E(i)$.

An *infinitesimal generator matrix* Q may be associated to a CTMC by setting the off-diagonal entries of the matrix Q with $q_{ij} = r_{ij}$, and the diagonal entries with $q_{ii} = -E(i)$. The matrix Q (or R) is usually sparse; further details about the properties of these matrices can be found in [Ste94]. We show a simple example of a CTMC in Figure 2.1. We use the example of a web server which accepts requests for TCP connections from arbitrary clients. Only one connection can be processed by the server at a time. Other requests arriving when the server is busy are enqueued in a single queue which can hold a maximum of 3 requests. The CTMC shown in Figure 2.1 models the queue of requests for the connection to the web server. The CTMC contains four states; state i indicates the number of connection requests in the queue. The arrival rate of requests is 2 and the service rate of the server is 3. Figure 2.1(b) shows the generator matrix Q associated to the CTMC. The transition rate matrix R for the CTMC can be obtained by replacing the diagonal entries of the generator matrix with zeros.



(a) A request queue modelled as a CTMC

$$\begin{pmatrix} -2 & 2 & 0 & 0 \\ 3 & -5 & 2 & 0 \\ 0 & 3 & -5 & 2 \\ 0 & 0 & 3 & -3 \end{pmatrix}$$

(b) The associated generator matrix (Q)

Figure 2.1: A CTMC and the associated generator matrix

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 will focus in this thesis on the steady-state solution of a CTMC. See the next section for more details.

Finally, we define the notion of reachability. We say that there exists a *transition* from state i to state j if $q_{ij} > 0$. A state j in a CTMC is reachable from another state i in the CTMC if there exists a finite sequence of transitions in the model from the state i to the state j. The set of all states which can be reached from the initial state is called the set of reachable states of the model.

2.1.2 Numerical Problem and Methods

Let $Q \in \mathbb{R}^{n \times n}$ be the infinitesimal generator matrix of a continuous time Markov chain with n states, and

$$\pi(t) = [\pi_0(t), \pi_1(t), \ldots, \pi_{n-1}(t)] \tag{2.3}$$

the transient state probability row vector, where $\pi_i(t)$ denotes the probability of the CTMC being in state i at time t. The transient behaviour of the CTMC is described by the Chapman-Kolmogrov differential equation:

$$\frac{d\pi(t)}{dt} = \pi(t)\, Q. \tag{2.4}$$

To compute this, the initial probability distribution of the CTMC, $\pi(0)$, is also required. We concentrate on the steady-state behaviour of a CTMC which is obtained by solving the system of equations:

$$\pi Q = 0, \ \sum_{i=0}^{n-1} \pi_i = 1, \tag{2.5}$$

where $\pi = \lim_{t\to\infty} \pi(t)$ is the *steady-state probability* vector. A sufficient condition for the unique solution of the Equation (2.5) is that the CTMC is finite and irreducible. A CTMC is *irreducible* if every state can be reached from every other state [Ste94]. In this thesis, we restrict our attention to solving only irreducible CTMCs; for details on the solution in the general case, see [Ste94], for example. The Equation (2.5) can be reformulated as $Q^T\pi^T = 0$, and well-known methods for the solution of systems of linear equations of the form $Ax = b$ can be used.

Using Equation (2.5), the steady-state probabilities for the CTMC shown in Figure 2.1 can be computed by solving the following system of linear equations:

$$\begin{aligned}
-2\pi_0 + 3\pi_1 &= 0,\\
2\pi_0 - 5\pi_1 + 3\pi_2 &= 0,\\
2\pi_1 - 5\pi_2 + 3\pi_3 &= 0,\\
2\pi_2 - 3\pi_3 &= 0,\\
\pi_0 + \pi_1 + \pi_2 + \pi_3 &= 1,
\end{aligned}$$

which yields the steady-state probability vector $\pi = [\frac{27}{65}, \frac{18}{65}, \frac{12}{65}, \frac{8}{65}]$.

Numerical Methods

The numerical solution methods for linear systems of the form $Ax = b$ are broadly classified into two categories: *direct methods*, such as *Gaussian elimination*, *LU factorisation* etc; and *iterative methods*. Direct methods obtain the exact solution in finitely many operations and are often preferred to iterative methods in real applications because of their robustness and predictable behaviour. However, as the size of the systems to be solved increases, they often become almost impractical due to the phenomenon known as *fill-in*. The fill-in of a sparse matrix is a result of those entries which change from an initial value of zero to a nonzero value during the factorisation phase, e.g. when a row of a sparse matrix is subtracted from another row, some of the zero entries in the latter row may become nonzero. Such modifications to the matrix mean that the data structure employed to store the sparse matrix must be updated during the execution of the algorithm.

Iterative methods, on the other hand, do not modify matrix A; rather, they involve the matrix only in the context of matrix-vector product (MVP) operations. 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]. Beginning with a given approximate solution, these methods modify the components of the approximation, until convergence is achieved. They do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods. They can be further classified into *stationary* methods like *Jacobi* and *Gauss-Seidel* (GS), and *non-stationary* methods such as *Conjugate Gradient*, *Lanczos*, etc. The volume of literature available on iterative methods is huge, see [BBC+94, Axe96, GL96, GO93, Saa03]. In [SV00], Saad and Vorst present a survey of the iterative methods; [Ste94] describes iterative methods in the context of solving Markov chains.

2.2 Basic Iterative Methods

In this section, we consider the so-called *stationary* iterative methods, those which can be expressed in the simple form $x^{(k)} = Fx^{(k-1)} + c$, where $x^{(k)}$ is the approximation to the solution vector at the k-th iteration and neither F nor c depend on k [BBC+94].

2.2.1 Power Method

Before we consider the iterative methods for the solution of the system of equations $Ax = b$, we would like to mention the *Power method* for the steady-state solution of Equation (2.5). Given the generator matrix Q, setting $P = I + Q/\alpha$ in Equation (2.5), where $\alpha \geq max_i \mid q_{ii} \mid$, leads to:

$$\pi P = \pi, \tag{2.6}$$

where π is the steady-state probability vector. In fact, for good convergence, the parameter α should be chosen very close to the bound yet satisfying $\alpha > max_i \mid q_{ii} \mid$.

Using $\pi^{(0)}$ as the initial estimate, an approximation of the steady-state probability vector after k transitions is given by $\pi^{(k)} = \pi^{(k-1)}P$. The method successively multiplies the steady-state probability vector with the matrix P until convergence is reached.

The Power method is guaranteed to converge (for irreducible CTMCs). However, in practice, it takes a very long time to converge; below we consider alternative iterative methods which, although not guaranteed to converge, in practice converge much faster than the Power method.

2.2.2 Jacobi Method

We now consider the iterative methods for the solution of the system of equations $Ax = b$, where A is of size n. In the k-th iteration of the Jacobi method, we calculate:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i \; - \; \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right), \tag{2.7}$$

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

$$x^{(k)} = D^{-1}(L + U)\, x^{(k-1)} \; + \; D^{-1}b, \tag{2.8}$$

1. `while` not converged
2. $\tilde{x} \leftarrow b - \check{A}x$
3. $\tilde{x} \leftarrow D^{-1}\tilde{x}$
4. Test for convergence
5. $x \leftarrow \tilde{x}$
6. `end while`

Figure 2.2: A standard Jacobi algorithm

where $A = D - (L + U)$ is a partitioning of A into its diagonal, lower-triangular and upper-triangular parts, respectively. Note the similarities between $x^{(k)} = Fx^{(k-1)} + c$ and Equation (2.8) above. The Jacobi method can be formulated into an MVP (*matrix-vector product*) based algorithm, for example, as shown in Figure 2.2.

In Figure 2.2, $\check{A}$ contains the off-diagonal elements of matrix A, i.e. $\check{A} = -(L + U)$. Observe also the similarities between the algorithm and Equation (2.8) above. Line 2 of the algorithm performs the MVP operation. The algorithm requires storage for two iteration vectors (the previous iterate x and the new iterate $\tilde{x}$), for the matrix $\check{A}$ and for the diagonal entries in D. Note that the new approximation of the solution vector is calculated using only the old approximation of the solution. This makes the Jacobi method well suited for parallelisation, but means it tends to exhibit slow convergence.

2.2.3 Gauss-Seidel Method

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

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)} \right) \tag{2.9}$$

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

$$x^{(k)} = (D - L)^{-1} U \, x^{(k-1)} + (D - L)^{-1} \, b \tag{2.10}$$

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

2.2.4 Successive Over-Relaxation (SOR) Method

An SOR iteration is given by:

$$x_i^{(k)} = \omega \hat{x}_i^{(k)} + (1-\omega) x_i^{(k-1)}, \quad 0 \leq i < n \tag{2.11}$$

where $\hat{x}$ denotes a GS iterate, and $\omega \in (0,2)$ is the 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 GS. 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 GS. However, a priori computation of an optimal value for ω is not feasible.

2.3 Block Iterative Methods

Consider a partitioning of the state space S of a CTMC into P contiguous partitions $S_0, \ldots, S_{P-1}$ of sizes $n_0, \ldots, n_{P-1}$, such that $n = \sum_{i=0}^{P-1} n_i$. Using this, the matrix A can be divided into P^2 blocks, $\{A_{ij} \mid 0 \leq i,j < P\}$, 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 $n_i \times n_j$. Using such a partitioning of the state space for $P = 4$, the system of equations $Ax = b$ can be partitioned as:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} \tag{2.12}$$

Given the partitioning introduced above, block iterative methods essentially involve the solution of P sub-systems of linear equations, of sizes $n_0, \ldots, n_{P-1}$ within a *global* iterative structure, say Gauss-Seidel; hence the block Gauss-Seidel method. From Equations (2.9) and (2.12), the *block Gauss-Seidel* method for the solution of the system $Ax = b$ is given by:

$$A_{ii}\ X_i^{(k)} = B_i - \sum_{j<i} A_{ij}\ X_j^{(k)} - \sum_{j>i} A_{ij}\ X_j^{(k-1)}, \quad 0 \leq i < P \tag{2.13}$$

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. Hence, in each of the P phases of the k-th iteration of the block Gauss-Seidel iterative method, we solve the Equation (2.13) for $X_i^{(k)}$. These subsystems 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; for further details, see [Ste94, DS00], for example.

2.4 Krylov Subspace Methods

We briefly mention Krylov subspace methods in this section. We have not employed these methods for the implementations of the solution techniques presented in this thesis. However, these methods are mentioned here because of their importance.

The *Krylov subspace methods* are iterative methods for the solution of large linear systems of the form $Ax = b$ that offer faster convergence than the methods discussed in the previous sections and do not require *a priori* estimation of parameters depending on the inner properties of the matrix. Furthermore, they are based on matrix-vector product computations and independent vector updates, which makes them particularly attractive for parallel implementations. Krylov subspace methods for arbitrary matrices, however, require multiple iteration vectors which makes it difficult to apply them to the solution of large systems of linear equations.

Nearly all notable Krylov subspace methods have been used to solve Markov models. The sizes of the models solved, however, are relatively small. Philippe et al. [PSS92] use various numerical methods including the

GMRES method, and report results for Markov models with up to 24,000 states. Buchholz [Buc99a, Buc99b] experimented (in addition to the stationary iterative methods) with GMRES, Arnoldi, conjugate gradient squared, BCGStab, and TFQMR methods, and reported results for models with up to 0.5 million states. In [Buc00], Buchholz employed projection methods in a multilevel setting and reported solutions for structured Markov chains with approximately one million states.

The *conjugate gradient squared* (CGS) method [Son89], to our knowledge, is the only Krylov subspace method used in the past for solving large Markov models. It performs 2 MVPs, 6 vector updates and two vector inner products during each iteration, and requires 7 iteration vectors. Further discussion of the Krylov methods is beyond the scope of this thesis. For details on Krylov subspace methods for the solution of Markov chains, see [Saa91, Saa95, DS00], for instance. A CGS algorithm for the steady-state analysis of CTMCs can be found in [KH99]; an interested reader may also find the paper [Meh03] relevant, where a survey of the Krylov subspace methods is included. In addition, the paper [SV00] is an excellent account of iterative methods.

2.5 Test for Convergence

Consider ξ is the *residual* vector, $b - Ax$, for the linear equation system $Ax = b$. Consequently, $\xi = 0$ for the desired solution x of the system. An iterative algorithm is said to have converged after k iterations if the magnitude of the *residual* vector becomes zero or desirably small. Usually, a test is carried out in each iteration to test for convergence; [BBC+94] discusses this subject in some detail. A frequent choice for the convergence test is to compare the *Euclidean norm* (also known as the l_2-*norm*) of $\xi^{(k)}$ (residual vector in the k-th iteration), calculated as:

$$\|\xi^{(k)}\|_2 = \sqrt{\xi^{(k)^T}\xi^{(k)}}, \tag{2.14}$$

with some predetermined threshold, usually $\epsilon\|\xi^0\|_2$ for $0 < \epsilon \ll 1$; $\|\xi^0\|_2$ denotes the *Euclidean norm* of the initial *residual* ξ^0, calculated as $b - Ax^0$. Another convergence criterion used in the iterative solution of Markov chains

is to check the l_∞-*norm*:

$$\|x^{(k)} - x^{(k-1)}\|_\infty = \max_i \mid x_i^{(k)} - x_i^{(k-1)} \mid, \tag{2.15}$$

until it falls below some ϵ ($0 < \epsilon \ll 1$). In the context of the steady-state solution of a CTMC, a widely used convergence criterion is the so-called relative (l_∞-norm) error criterion:

$$\max_i \left(\frac{\mid x_i^{(k)} - x_i^{(k-1)} \mid}{\mid x_i^{(k)} \mid} \right) \; < \; \epsilon \; \ll 1. \tag{2.16}$$

It is found to have less erratic behaviour than the other criteria mentioned above.

2.6 Sparse Matrix Storage

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. Our discussion of sparse schemes in this section is 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 2.3 gives a 4×4 sparse matrix with $a = 6$ 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 2.4(a) illustrates the storage of the matrix in Figure 2.3(a) in the *compressed sparse row* (CSR) [Saa90] format. All the $a + n$ nonzero entries

$$n = 4 \quad a = 6 \qquad \begin{pmatrix} -0.2 & 0 & 0.2 & 0 \\ 0 & -0.9 & 0.4 & 0.5 \\ 0.6 & 0.7 & -1.3 & 0 \\ 0.9 & 0 & 0 & -0.9 \end{pmatrix}$$

(a) A CTMC generator matrix

Val	−0.2	0.2	−0.9	0.4	0.5	0.9	0.7	−1.3	0.6	−0.9
Col	0	2	1	2	3	0	1	2	0	3
Row	0	0	1	1	1	3	2	2	2	3

(b) The coordinate format

Figure 2.3: A 4×4 sparse matrix and its storage in the coordinate format

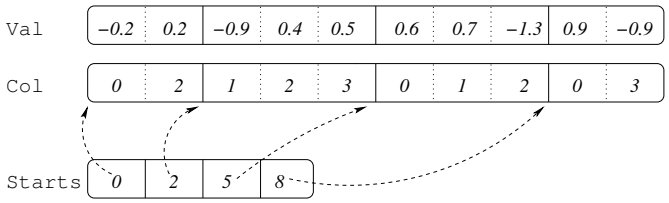
are stored row by row in the array `Val`, while `Col` contains column indices of these nonzero entries; the elements within a row can be stored in any order. The i-th element of the array `Starts` (of size n ints) contains the index in `Val` (and `Col`) of the beginning of the i-th row. The CSR format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

2.6.1 Separate Diagonal Storage

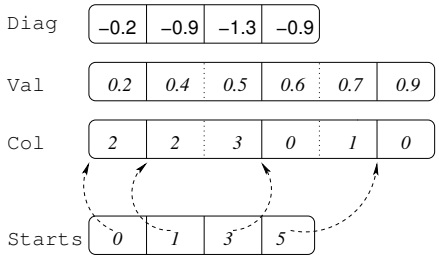
A second dimension for sparse schemes is added if we consider that many iterative algorithms treat diagonal entries of a matrix differently. This gives us the following two additional choices for sparse storage schemes; an alternative choice will be mentioned in Section 2.6.2.

The Modified Sparse Row Format

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



(a) The compressed sparse row format



(b) The modified sparse row format

Figure 2.4: The CSR and MSR formats for the example matrix

is known as the *modified sparse row* (MSR) format [Saa90] (a modification of CSR), see Figure 2.4(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.

Avoiding the Diagonal Storage

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 during the iterative solution phase. We define the matrix D as the diagonal matrix with $d_{ii} = q_{ii}$, for $0 \leq i < n$. 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 $y = D\pi^T$. 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 for each step in an iterative method such as Gauss-Seidel.

2.6.2 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 $MaxD \leq 2^{16}$; 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 various names. We call it the *indexed MSR* format.

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* is less than 2^{16}, or 2^8, respectively. In fact, for each index, $\lceil log_2(MaxD) \rceil$ bits suffice. Similarly, it is also possible to index diagonal entries of a matrix provided the diagonal vector has relatively few

k	states (n)	off-diagonal nonzero (a)	a/n	memory required for Q (MB)		MB per π
				MSR format	*Indexed MSR*	
8	4,459,455	38,533,968	8.64	489	234	34
9	11,058,190	99,075,405	8.96	1,260	598	84
10	25,397,658	234,523,289	9.23	2,962	1,415	194
11	54,682,992	518,030,370	9.47	6,554	3,120	417
12	111,414,940	1,078,917,632	9.68	13,563	6,492	850
13	216,427,680	2,136,215,172	9.87	26,923	12,842	1,651
14	403,259,040	4,980,958,020	12.35	61,609	29,652	3,077
15	724,284,864	9,134,355,680	12.61	112,810	54,334	5,526

Table 2.1: Comparison of MSR and indexed MSR sparse formats

distinct entries. This justifies an alternative choice for separate diagonal storage (see Section 2.6.1).

Table 2.1 gives a fact sheet for a model of a flexible manufacturing system (FMS) [CT93] comparing storage requirements in MSR and indexed MSR formats. The first column in the table gives the model parameter k (see Appendix B); 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 fourth column (a/n) gives the average number of the off-diagonal nonzero entries per row, an indication of the matrix sparsity. Columns 5 and 6 give the storage requirements for matrices (including storage for the diagonal) in MB for the MSR and indexed 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. The largest model reported in the table is FMS($k = 15$) with over 724 million states and 9 billion transitions.

2.7 Implicit Matrix Storage

In the previous section, we have discussed various schemes to store a generator matrix explicitly. Another approach, which has been enormously successful in model checking, is implicit storage of the matrix. These techniques are usually known as "implicit" because they do not require data structures of size proportional to the number of states. These methods, which can be

traced back to Binary Decision Diagrams (BDDs) [Bry86] and the Kronecker approach [Pla85], include multi-terminal binary decision diagrams, Matrix Diagrams (MD) [CM99, Min01] and on-the-fly methods [DS98b]. In the following sections, we describe MTBDDs, which we will use later in the thesis. The interested reader may follow the individual references for further details on each of the implicit methods; see also [MP04, BK04] for recent surveys of such data structures.

2.7.1 Multi-Terminal Binary Decision Diagrams

Multi-Terminal Binary Decision Diagrams (MTBDDs) [CFM+93, BFG+93] are a simple extension of binary decision diagrams (BDDs). An MTBDD is a rooted, directed acyclic graph (DAG), which represents a function mapping Boolean variables to real numbers. MTBDDs can be used to encode real-valued vectors and matrices by encoding their indices as Boolean variables. Since a CTMC is described by a square, real-valued matrix, it can also be represented as an MTBDD. Techniques which use data structures based on BDDs are often called *symbolic* approaches.

The advantage of using MTBDDs (and other symbolic data structures) to store CTMCs is that they can often provide extremely compact storage, provided that the CTMCs exhibit a certain degree of structure and regularity. In practice, this is very often the case since they will have been specified in some, inherently structured, high-level description formalism. Intuitively, the reason that MTBDDs can exploit such regularity is that the data structure is stored in a reduced form, with identical nodes of the graph being merged. This means that, where possible, identical portions of the matrix are stored only once.

In the work in this thesis, we have actually used a variant of MTBDDs called *offset-labelled MTBDDs* [Par02, KNP02b, KNP04b]. The principal difference is the addition of *offsets* to each node of the graph, used to allow conversion between the *potential* and *actual* (reachable) state spaces. The potential state space is often significantly larger than the actual state space and hence inefficient to deal with. The reason for selecting offset-labelled MTBDDs will be clarified further in Section 2.7.3.

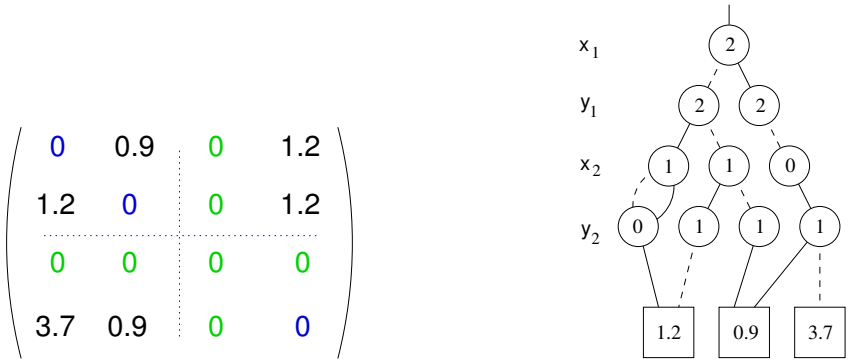
2.7.2 Offset-Labelled MTBDDs

In this section, we briefly describe the offset-labelled MTBDD data structure. A detailed account of the data structure can be found in [Par02]. Occasionally, in this thesis, we may refer to this data structure as an MTBDD.

To describe MTBDDs, we use a similar matrix as used for the explanation of sparse storage schemes in Section 2.6. For the sake of explanation, however, we make some changes in the actual values of the matrix in Figure 2.3(a). Figure 2.5 shows a matrix which might occur in the numerical solution of a CTMC, its representation as an offset-labelled MTBDD, and a table explaining how the information is encoded. Note that there are six off-diagonal entries in the matrix; only three of these are distinct (0.9, 1.2, and 3.7). Note also that state 2 of the CTMC is unreachable. 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 entries of the matrix in Figure 2.5 are all zero.

The offset-labelled MTBDDs comprise two types of nodes: *non-terminal* nodes, drawn as circles, and *terminal* nodes, drawn as squares. Non-terminal nodes are labelled with integer offsets, terminal nodes with real values. Each row of nodes in the data structure is associated with a Boolean variable which is written on the far left of the row. The MTBDD represents a function over these Boolean variables. For the example in Figure 2.5, the function is over the variables $\mathsf{x}_1, \mathsf{y}_1, \mathsf{x}_2, \mathsf{y}_2$. For a given valuation of these variables, the value of the function can be computed by tracing a path from the top of the MTBDD to the bottom, at each node taking the *dotted edge* if the associated Boolean variable is 0 and the *solid edge* if it is 1. The MTBDD is ordered such that on each path from the root to a terminal node, the variables are visited in the same order. The function value can be read from the label of the terminal node reached. For example, if $(\mathsf{x}_1, \mathsf{y}_1, \mathsf{x}_2, \mathsf{y}_2) = (1, 0, 1, 1)$, the function returns 0.9. If there is no such path, the value is 0. As in Figure 2.5(a), for clarity, the zero terminal node of an MTBDD and any edges leading directly to this node are usually not drawn. The functions $(\mathsf{x}_1, \mathsf{y}_1, \mathsf{x}_2, \mathsf{y}_2) = (0, 0, 0, 0)$ and $(\mathsf{x}_1, \mathsf{y}_1, \mathsf{x}_2, \mathsf{y}_2) = (1, 1, 0, 0)$ both return zero. We indicate this in the figure, by listing "-" in the corresponding "Path" entry for the first and the last row in the table. For explanation, we (only) list two zero entries in the table.

To represent the matrix, the offset-labelled MTBDD in Figure 2.5 uses



(a) A CTMC and its offset-labelled MTBDD representation

Matrix entry	Encoding				Path				Offsets				Reachable entry
	x_1	x_2	y_1	y_2	x_1	y_1	x_2	y_2	x_1	y_1	x_2	y_2	
$(0,0) = 0$	0	0	0	0	0	0	0	-	-	-	-	-	$(0,0) = 0$
$(0,1) = 0.9$	0	0	0	1	0	0	0	1	-	-	-	1	$(0,1) = 0.9$
$(0,3) = 1.2$	0	0	1	1	0	1	0	1	-	2	-	0	$(0,2) = 1.2$
$(1,0) = 1.2$	0	1	0	0	0	0	1	0	-	-	1	-	$(1,0) = 1.2$
$(1,3) = 1.2$	0	1	1	1	0	1	1	1	-	2	1	0	$(1,2) = 1.2$
$(3,0) = 3.7$	1	1	0	0	1	0	1	0	2	-	0	-	$(2,0) = 3.7$
$(3,1) = 0.9$	1	1	0	1	1	0	1	1	2	-	0	1	$(2,1) = 0.9$
$(3,2) = 0$	1	1	1	0	1	-	-	-	-	-	-	-	$(2,2) = 0$

(b) Encoding of the offset-labelled MTBDD given in (a) above

Figure 2.5: A CTMC matrix and its offset-labelled MTBDD representation

the variables x_1, x_2 to encode row indices and y_1, y_2 to encode column indices. We call the nodes associated with the Boolean variables x_i, the *row nodes*, and the nodes associated with the variables y_i, the *column nodes*. 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.

Each non-terminal node of an offset-labelled MTBDD represents a submatrix of the matrix represented by the whole MTBDD. Since an MTBDD is based on binary decisions, each x_i node divides the (sub)matrix it represents into two submatrices, i.e. the top submatrix (from the dotted edge) and the bottom submatrix (from the solid edge). Similarly, each y_i node divides the matrix into the left submatrix and the right submatrix. We define a *level* of an MTBDD to be an adjacent pair of rows of nodes corresponding to a matching pair of variables x_i and y_i; counting levels from the top of the MTBDD, level i contains all the x_i and y_i nodes. As a consequence, descending each level of the data structure splits the matrix into 4 submatrices. In Figure 2.5, for example, each of the three x_2 nodes represents a 2×2 submatrix, of the 4×4 matrix.

As noted earlier, the distinguishing feature of the offset-labelled MTBDDs is that the MTBDD nodes are labelled with integer values called *offsets*. These offsets are used to compute the *actual* row and column indices of the matrix entries. Actual indices here means the indices in terms of reachable states only. As mentioned earlier, this is typically important since the potential state space can be much larger than the actual state space (for examples, see Table 4.2 on Page 81). Essentially, for a row node, the offset denotes the number of reachable rows in the top submatrix, and for a column node, the offset denotes the number of reachable columns in the left submatrix. 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 using y_i nodes. In the CTMC example in Figure 2.5, state 2 is not

reachable (it has no incoming transitions, i.e. column 2 of the matrix contains only zero entries). For the previous example of matrix entry (3,1), the actual row index is 2+0=2 and the column index is 1, i.e. the matrix entry (3,1) in the potential state space corresponds to the entry (2,1) in the actual state space.

For efficiency reasons, pure MTBDDs are usually stored in a reduced form. The *identical* nodes in an MTBDD, i.e nodes which share the same level and have identical children, are merged. The *don't care* nodes in the MTBDD, i.e. the nodes for which both the dotted and solid edges point to the same node, are removed or *skipped*. In an offset-labelled MTBDD, however, nodes cannot be skipped; the don't care nodes which point to the zero terminal node are an exception. The reason for this restriction is that the nodes are required on each level so that the nodes can be labelled with the respective offsets. Secondly, in an offset-labelled MTBDD, we have the additional restriction that merging of nodes can only occur if the nodes are identical in terms of their level, children and offset labellings.

Finally, we briefly describe the construction of an offset-labelled MTBDD from a pure MTBDD; for a detailed description of this process, see [Par02]. Firstly, a (reduced) BDD representation of the reachable state space is modified by reinserting the don't care nodes, and by labelling the BDD nodes with appropriate offsets. More precisely, this BDD over k Boolean variables, $\mathsf{x}_1, \mathsf{x}_2, \cdots \mathsf{x}_k$, encodes a vector of length 2^k. The vector corresponds to the state space of a CTMC. The values in the vector are Boolean, i.e. the values belong to the set $\mathbb{B}$. A state is reachable if the corresponding entry in the vector is 1, and the states associated to the zero vector entries are unreachable. Reinserting the don't care nodes to this reduced BDD is easy; the edges which skip one or more nodes can be checked for, and where appropriate, new nodes are inserted. In analogy to an MTBDD, each non-terminal node of a BDD divides the vector it represents into two subvectors, the top and the bottom subvectors. An offset, in this case, denotes the number of reachable states in the top subvector. The offset value for a node can be computed by recursively adding the number of reachable states of its children and grandchildren.

Once an offset-labelled BDD has been built, it is used to construct an offset-labelled MTBDD. Firstly, as for the BDD, the don't care nodes are

reinserted into a pure MTBDD. This can be carried out in a similar manner. Secondly, each node of this MTBDD is labelled with appropriate offsets. This is accomplished with the help of two offset-labelled BDDs, one for the row nodes and one for the column nodes. The MTBDD and the two instances of the offset-labelled BDD are concurrently traversed during this process. Possible conflicts of offsets due to the merged nodes are also checked for and, where applicable, additional offset-labelled nodes are inserted.

2.7.3 Numerical Solution with MTBDDs

Numerical solution of CTMCs can be performed purely using conventional MTBDDs; see for example [HMPS96, HMKS99, HKN+03]. This is done by representing both the matrix and the vector as MTBDDs and using an MTBDD-based matrix-vector multiplication algorithm (see [CFM+93, BFG+93] for instance). However, this approach is often very inefficient because, during the numerical solution phase, the solution vector becomes more and more irregular and so its MTBDD representation grows quickly.

A second disadvantage of the purely MTBDD-based approach is that it is not well suited to the Gauss-Seidel iterative method, only the Jacobi method. In fact, an MTBDD version of Gauss-Seidel, using matrix-vector multiplication, has been presented in the literature [HMKS99]. However, this relies on computing and representing matrix inverses using MTBDDs. Generally, this will be inefficient because converting a matrix to its inverse will usually result in a loss of structure and possibly fill-in. As mentioned in Section 2.2, an efficient implementation of Gauss-Seidel is desirable because it typically converges faster than Jacobi and it requires only one iteration vector instead of two.

Fortunately, significant progress has been made to address the problems of the purely MTBDD-based approach mentioned above. This is realised in offset-labelled MTBDDs [KNP04b, Par02], described earlier in Section 2.7.2. The first restriction of pure MTBDDs regarding the vector representation is resolved by combining MTBDD-based storage (with addition of offsets to the MTBDD nodes) of the matrix with explicit, array-based storage of the solution vector. A matrix-vector multiplication operation, as required in an iteration of numerical solution, can now be performed by a single depth-first traversal of the data structure. This is because a multiplication needs

access to every matrix element exactly once and each element corresponds to a path through the MTBDD. Unsurprisingly, the overhead associated with this process makes an iteration of iterative method slower than the equivalent operation with sparse matrices since it is significantly faster to read the matrix entries directly from an array-based data structure. This, however, can be addressed as follows.

We have seen in the previous section that MTBDDs provide a natural decomposition of a matrix into its submatrices. An MTBDD is traversed recursively, from one node to another. Each node in the MTBDD represents a (sub)matrix, divides this submatrix into its two submatrices, and provides access to these submatrices through its two edges. The nodes near the bottom of the MTBDD, in particular, are visited many times during its traversal. It is much faster to extract entries of the matrix if some of these nodes can be replaced with an explicit representation of their corresponding submatrix, eliminating the need to traverse the nodes below this point. With these modifications to the approach, it has been shown in [KNP04b, Par02] that the resulting data structure furnishes significantly faster numerical solutions while retaining the symbolic storage advantages. Numerical solution using offset-labelled MTBDDs has also been applied successfully in [KSW04].

One remaining drawback of offset-labelled MTBDD-based numerical solution is that, although the Jacobi iterative method can be efficiently implemented, Gauss-Seidel cannot because it requires row-wise access to matrix entries. A depth-first traversal of the MTBDD does not allow matrix entries to be extracted in this order. Of course, it would be possible to access each element of each row individually, going from top to bottom of the MTBDD each time, but this would be very inefficient.

This limitation can be relaxed to some extent by making use of the fact that MTBDDs allow convenient access to matrix blocks. Descending one level from the top of an MTBDD splits the matrix which it represents into 4 blocks, and, therefore descending l levels, gives a decomposition into $(2^l)^2$ blocks. If pointers to the nodes representing these blocks are stored in an array-based data structure, these matrix blocks can be accessed swiftly without having to traverse the top part of the MTBDD. This allows efficient access to each row of matrix blocks, and therefore, implementation of block iterative methods (see Section 2.3) with MTBDDs is possible. Based on this optimisation, the

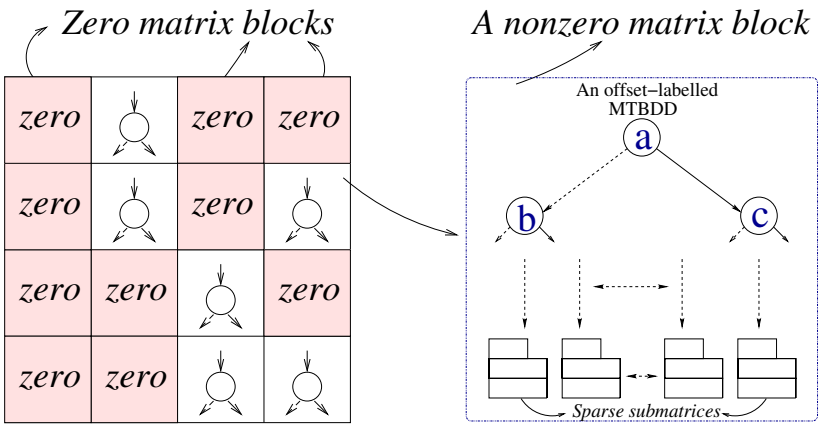


Figure 2.6: A matrix as an offset-labelled MTBDD: another perspective

pseudo Gauss-Seidel iterative method was introduced in [Par02] which typically converges faster than the Jacobi iterative method and requires storage of only one iteration vector. We give a pseudo Gauss-Seidel algorithm in Chapter 4, where we have employed this method to implement our symbolic out-of-core method.

Finally, in Figure 2.6, we give the reader an intuition of the offset-labelled MTBDDs. The figure depicts a simplified perspective of the data structure. The matrix in the figure is shown to be decomposed into a number of blocks. Most of these blocks are zero, i.e. all entries in these blocks are zero. On the top layer, the offset-labelled MTBDD data structure stores a sparse matrix, each nonzero entry thereof is a *reference* to an offset-labelled MTBDD node. For illustrative reason, in the figure, each such reference is shown as a node with an incoming and two outgoing edges. The top-layer matrix is stored using the CSR sparse format (see Section 2.6). Each nonzero entry of this matrix, i.e. a block, itself is stored as an offset-labelled MTBDD. A nonzero matrix block represented as an offset-labelled MTBDD is shown on the right side of figure. The top three nodes of the offset-labelled MTBDD (labelled with offsets a, b, and c) are drawn. The bottom part of the offset-labelled MTBDD is stored using the explicit sparse representation of the bottom submatrices. Further explanation of offset-labelled MTBDDs follows in Chapter 5, where we introduce modifications to the data structure.

2.8 Related Work

Recall that a large variety of useful performance measures can be derived from a CTMC via the computation of its steady-state probabilities. This reduces to the more general problem of solving a linear equation system $Ax = b$ of size equal to the number of states in the CTMC. Unfortunately, these models tend to grow extremely large, as we know, due to a phenomenon known as the state space explosion problem.

A great deal of effort has gone into developing efficient implementations for the computation of steady-state probabilities of CTMCs, with particular emphasis on the problem of storing large CTMCs. This is also the focus of our research. As mentioned earlier, the various approaches can be broadly classified into, firstly, *explicit* approaches, where the CTMC is kept in a data structure of size proportional to the number of states and transitions, typically using a sparse matrix scheme, or, secondly, *implicit* approaches, where this explicit storage is avoided. Explicit approaches typically provide faster solutions due to the fast, array-based data structures used (Section 2.6). Implicit techniques offer compact storage by exploiting structure and regularity in the CTMCs, and can hence be applied to larger CTMCs than explicit methods. Another classification of the solution techniques is into *in-core* approaches, where data is stored in the main memory of a computer, and *out-of-core* approaches, where it is stored on disk.

Another direction taken to combat the state space explosion problem is to use the large amount of memory and compute power available with parallel and distributed computers. Therefore, the solution techniques are either developed for single workstations, i.e. *serial*, or for parallel architectures. Parallel solution techniques can be further divided into three categories. These include the standard parallel approaches where CTMC (in a sparse format) is stored using only RAM of the parallel machine, secondly, the parallel techniques which are based on the implicit CTMC storage, and finally, the parallel techniques which rely on the out-of-core storage of CTMCs.

This section presents a survey of the work related to this thesis, i.e. efficient storage and numerical methods for steady-state analysis of large CTMCs. The techniques to combat the state space problem are summarised in the next two subsections; Section 2.8.1 reviews the techniques developed for a single workstation, and Section 2.8.2 considers parallel architectures.

2.8.1 State Space Explosion: Serial Solutions

In this section, we review the main representative approaches used for overcoming the state space explosion problem when analysing stochastic models. We mention only those techniques which are targeted for single workstations. We outline these approaches in Table 2.2, and concentrate on the iterative methods employed, the data structures used to store the matrix and vector, and whether they are stored in-core or out-of-core. The approaches listed in the table are sorted into implicit and explicit categories. In the table, we also include the work presented in this thesis in order to give a clear picture of how our work relates to the existing work in the literature. We first consider the implicit methods.

Multi-Terminal Binary Decision Diagrams [CFM+93, BFG+93] provide a compact representation for CTMC matrices. However, the pure MTBDD approach, where both the matrix as well as the vector are stored using MTBDDs [HMPS96, HMKS99, HKN+03], as mentioned earlier, is often impractical for large models. This is due to an inefficient MTBDD representation of the probability vector which grows larger through loss of structure and rapidly increasing number of distinct values. Secondly, the pure MTBDD approach is not well suited to the Gauss-Seidel iterative method which typically converges faster than Jacobi and requires only one iteration vector instead of two.

The offset-labelled MTBDD method [KNP04b, Par02] combines MTBDDs and explicit techniques. The matrix is stored using an MTBDD, labelled with offsets, and the vector is stored in-core, as an array. We know from Section 2.7.3 that this approach can be used with the pseudo Gauss-Seidel (PGS) iterative method. The method typically converges faster than the Jacobi method and has relatively low memory requirements compared to Jacobi. Its limitations are in the explicit in-core storage of the iteration vector, and in that it does not permit an efficient implementation of the standard Gauss-Seidel iterative method. The offset-labelled MTBDDs have also been successfully used for the numerical solution in [KSW04].

The next three approaches also rely on exploiting structure in CTMCs. They use Kronecker algebra to derive implicit methods for the analysis of CTMCs. The philosophy behind the Kronecker approaches is that the CTMC matrix can be expressed as a set of smaller matrices which are com-

bined by Kronecker (tensor) operations. These submatrices correspond to the system components. This results in a very compact representation of the CTMC matrix. In [Pla85], Plateau first proposed the Kronecker approach as an efficient means for representing CTMC matrices of a formalism called stochastic automata networks (SANs). Several improvements over the original approach of Plateau have been reported in, for example, [Don94, Kem96, CT96, CT97, BCDK00, FPS98, UD98]. For instance, the original approach operates over the potential state space, representing all possible interleavings of the state spaces of the components. This requires storage for considerably larger solution vectors compared to the case if only the vector for reachable states is stored. This deficiency was removed [Kem96, CT96, CT97, BCDK00]. The original approach was also extended to certain other formalisms. Furthermore, efficient implementations of more advanced iterative methods including the Gauss-Seidel iterative method were realised [Ste94, BCDK00, UD98]. Consequently, these extensions allowed analysis of CTMCs at competitive speeds while maintaining the compact CTMC representation.

The next approach [CM99, Min00, Min01] is the work of Ciardo and Miner. They developed the matrix diagrams (MDs) data structure for efficient Kronecker-based solution of CTMCs. Like MTBDDs, a matrix diagram is a directed acyclic graph. In contrast to MTBDDs, the nodes for the matrix diagrams, however, are the submatrices which make up the Kronecker representation of the model. Matrix diagrams provide a compact representation for the storage of CTMCs. Furthermore, the authors have developed efficient implementations of the Gauss-Seidel iterative method. Solution of models with as many as 41 million states have been reported using their approach.

The probabilistic decision graphs (PDGs) data structure was originally introduced by Bozga and Maler in [BM99]. A PDG is a BDD-based data structure designed specifically for vectors and matrices of probabilities. The nodes in PDGs are labelled with conditional probabilities. A vector entry, for example, can be determined by multiplying the conditional probabilities along the corresponding path in the graph. The motivation for the PDG data structure was to provide a compact representation of matrices and vectors which cannot be stored compactly in MTBDDs. This data structure was later

extended by Buchholz and Kemper [BK01]. Modifications to PDGs were made to allow more than two edges from each node. For numerical solution, the matrix was stored using a Kronecker representation and the vector was kept as a PDG. However, the overhead associated with the manipulation of PDGs in this case was found to be considerable, leading to unsatisfactory speeds for the numerical solution.

We now consider the methods based on the explicit storage for both matrix and vector. In Table 2.2, we do not include the conventional explicit approaches where the CTMC matrix and vector are kept in-core, and instead we consider the methods which employ out-of-core techniques for the solution of CTMCs. Out-of-core techniques for the analysis of Markov chains have emerged as an effective method of combating the state space explosion problem. In this context, Deavours and Sanders [DS97] were the first to use an out-of-core technique for the steady-state solution of Markov models. This is the first explicit method listed in the table. The authors used a block Gauss-Seidel iterative method for the numerical solution of CTMCs. They stored the matrix on disk and read blocks of matrix into RAM when required. The in-core storage of iteration vector in this case, however, was still prohibitive. The authors introduced their disk-based tool in 1997, and later, in 1998, improved their tool by storing the CTMC to disk in a compressed format [DS98a]. We will give more technical discussion of this method in Chapter 3, where we compare it with the complete out-of-core method.

The other notable manuscripts on the out-of-core solution of CTMCs are [KH99] and [BH01, Bel03]. Since the main emphasis of these is on the parallelisation of the matrix out-of-core approach of Deavours and Sanders, we discuss these in Section 2.8.2.

In summary, all the implicit and explicit approaches which we have reviewed in this section provide a compact representation for CTMCs, or otherwise provide a solution to the state space explosion problem by employing out-of-core storage. However, it is recognised that [Cia01b, KNP04b] a major hurdle for all these approaches has been the storage of iteration vector(s). The techniques presented in this thesis (see Table 2.2) provide a solution for both implicit and explicit methods through out-of-core storage of the iteration vector.

Solution Method	Iterative Method	Matrix (Q)		Vector (π)	
		Data structure	Storage	Data structure	Storage
Implicit Methods					
MTBDDs [HMPS96, HMKS99, HKN$^+$03]	Jacobi	MTBDD	In-core	MTBDD	In-core
Offset-labelled MTBDDs [KNP04b, Par02]	PGS	Offset-labelled MTBDD	In-core	Array	In-core
Kronecker [Pla85, FPS98, UD98, BCDK00]	GS	Kronecker Expression	In-core	Array	In-core
Matrix Diagrams [CM99, Min00, Min01]	GS	Matrix Diagram	In-core	Array	In-core
PDGs [BK01]	GS	Kronecker Expression	In-core	PDG	In-core
Symbolic Out-of-core *this thesis*	PGS	Offset-labelled MTBDD	In-core	Array	Out-of-core
Improved OL-MTBDDs *this thesis*	GS	Improved OL-MTBDD	In-core	Array	In-core Out-of-core
Explicit Methods					
Matrix Out-of-core [DS97, DS98a]	GS	Sparse matrix	Out-of-core	Array	In-core
Complete Out-of-core *this thesis*	GS	Sparse matrix	Out-of-core	Array	Out-of-core

Table 2.2: Solution methods for CTMC analysis

2.8.2 State Space Explosion: Parallel Solutions

Shared memory multiprocessors, distributed memory computers, workstation clusters and grids provide a natural way of dealing with the memory and computing power problems, i.e. the task can be effectively distributed to parallel processors with shared or distributed memories. The goal of using parallel computers is to obtain high performance. Achieving this goal is usually a challenge in many areas of scientific computing. Various performance issues and techniques trade off with one another, which makes parallel computing so challenging and interesting. The books [KGGK94, GO93, GL96] contain a wealth of information on parallel computing. [CSG99] is an excellent book on parallel computer architecture.

In the previous section, we concentrated on the solution methods for the steady-state analysis of Markov chains, where the target architecture was a single, contemporary workstation. We now survey the parallel and distributed CTMC analysis techniques. We will use the terms "parallel" and "distributed" to refer to the techniques developed for shared memory and distributed memory architectures, respectively.

Much work is available concerning the parallel and distributed numerical iterative solution of general systems of linear equations, see [BBC+94, DV99, Saa03], for instance. There are two major directions taken in the context of solutions of Markov chains: firstly, the approaches based on explicit storage of the matrix, and secondly, the parallel techniques which rely on implicit CTMC storage. We first consider the explicit parallel approaches.

By far the most promising generally applicable techniques for the steady-state solution of large Markov models are iterative solution methods where matrices are stored explicitly in sparse format. Parallel solutions for the largest Markov models are included in this category. The two methods we first mention in this category are based on parallelisation of the matrix out-of-core technique.

In 1999, Knottenbelt and Harrison [KH99] introduced the first distributed matrix out-of-core solution for large CTMCs on a 16-node (256MB RAM per node) distributed memory computer. This was an effort to utilise the concurrent power as well as the secondary memory of parallel processing nodes for the solution of CTMCs. The authors used the Jacobi and conjugate gradient squared (CGS) iterative methods for the numerical solution. We know from

Section 2.4 that the CGS method requires storage for 7 iteration vectors, in addition to the matrix storage. However, since not all of these 7 vectors are needed at the same time, the intermediate vectors can be written to disk and hence storage requirements of the algorithm can be reduced. Knottenbelt [Kno99] used a parallel out-of-core version of CGS with a reduced in-core storage of 3 iteration vectors. The author reported results for CTMCs with as many as 94 million states.

In 2001, Bell and Haverkort [BH01] used the distributed matrix out-of-core techniques to further extend the size of solvable models, this time on a workstation cluster. The authors used the conjugate gradient squared method to solve smaller models. In the implementation of the CGS method, they kept 5 iteration vectors in-core and stored two vectors out-of-core. The largest model they solved using this iterative method contained 54 million states. For larger models, they used the Jacobi iterative method and reported the solution of a 724 million state system. It is interesting to mention here that the solution took 16 days to converge on a 26-node (52 processors) cluster with 13GB RAM and 1040GB disk space. The largest solvable model in this case was limited by the storage required (in-core) for the two iteration vectors.

The two approaches mentioned above relied on explicit out-of-core storage of the CTMC matrices. The remaining approaches in this category do not rely on out-of-core storage and use RAM to store all the explicit data structures. These include the distributed solution of Marenzoni et al. [MCC97], the shared memory parallel implementation of Allmaier et al. [AKH97], the parallel CTMC solution of [MPS99] based on a block iterative method, and distributed approach of [BDKW03] on a workstation cluster. See also [Cia01a], where Ciardo gives a survey of explicit distributed techniques for CTMC analysis; the author excludes the approaches implemented on a shared memory architecture.

We now consider parallel steady-state solution methods which are based on implicit storage of CTMCs. The first to mention in this category is the work of Buchholz et al. [BFK99]. The authors reported parallelisation of the Kronecker methods on a cluster of workstations. A representation of the generator matrix provided by the Kronecker methods partitions naturally into a block structure. These matrix blocks hence can be easily assigned

to distributed processors. The authors used a block Jacobi method for the steady-state solution and reported results for models with up to 8 million states.

CHAPTER

THREE

Explicit Out-of-Core Solution Methods

In this chapter, we present the explicit out-of-core algorithms which we have developed during the PhD studies for the steady-state solution of CTMCs. We begin by introducing an in-core block Gauss-Seidel algorithm in the next section. Then we introduce and explain the compact MSR scheme, a contribution of this thesis, in Section 3.2. The scheme offers significant saving over other sparse storage schemes. For the out-of-core methods, these savings (disk and RAM) also result in smaller amount of disk I/O. It has been used to store CTMC matrices in all our experiments presented in this chapter. This storage scheme will also be used in our work presented in Chapter 5.

In Section 3.3, we briefly explore the general issues involved in the design and analysis of out-of-core algorithms. In Section 3.4, we present and explain a matrix out-of-core algorithm based on the work of Deavours and Sanders [DS98a]. In Section 3.5, we describe our complete out-of-core algorithm which relaxes storage limitations for the matrix out-of-core methods. The implementation details of the two out-of-core algorithms are given in the corresponding subsections. Then, in Section 3.6, we present experimental results for the solution methods and analyze them in detail. A discussion of the work related to the material presented in this chapter is given in Section 3.7.

3.1 An In-Core Block Gauss-Seidel Algorithm

Recall from Chapter 2 that, in this thesis, we focus on the computation of the steady-state probabilities of a CTMC, a problem which corresponds to solving the system of equations $\pi Q = 0$, where $\pi \in \mathbb{R}^n$ and $Q \in \mathbb{R}^{n \times n}$. We have reviewed block iterative methods for the solution of linear equation systems in Chapter 2. In this section, we present and explain an in-core block Gauss-Seidel algorithm for the solution of the system $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. We reproduce from Chapter 2, the block Gauss-Seidel equation for the solution of the system $Ax = 0$:

$$X_i^{(k)} = -A_{ii}^{-1} \left(\sum_{j<i} A_{ij}\, X_j^{(k)} + \sum_{j>i} A_{ij}\, X_j^{(k-1)} \right), \quad 0 \leq i < P. \tag{3.1}$$

The partitioning of matrix A and iteration vector x employed in the above equation has been explained in Section 2.3. The equation can be refined into a block Gauss-Seidel algorithm, shown in Figure 3.1. The algorithm, in addition to the storage for the matrix, requires an array x of size n (states in the CTMC) 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-matrix-vector products (sub-MVPs), $A_{ij}X_j$, i.e., the multiplication of a single matrix block by a single vector block. The number n_{max} denotes the size of the largest matrix and vector blocks. 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 P such arrays.

Each iteration of the algorithm can be seen as progressing in P phases. In the i-th phase, elements from the i-th block of the iteration vector are updated. Note that 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 $0 \leq j < P$. We have also illustrated this fact with the help of Figure 3.2 for the second phase ($i = 1$) with $P = 4$; the matrix and vector blocks used in this computation 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 3.1 performs a *unit of computation*, a sub-MVP, and accumulates these products. Line 8 corresponds to solving a system of equations, either by direct or iterative methods (see Section 2.3). We use the Gauss-Seidel iterative method to solve $A_{ii}X_i = \tilde{X}_i$, for X_i.

```
0. while not converged
1.   for i = 0 to P − 1
2.       X̃_i ← 0
3.       for j = 0 to P − 1
4.           if j ≠ i
5.               X̃_i = X̃_i − A_ij X_j
6.           end if
7.       end for
8.       Perform A_ii X_i = X̃_i
9.       Test for convergence
10.  end for
11. end while
```

Figure 3.1: A block Gauss-Seidel algorithm

More precisely, we apply the following computations to update each of the n_i elements of the i-th vector block, X_i:

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

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}).

In summary, therefore, the block iterative algorithm of Figure 3.1 applies one Gauss-Seidel (inner) iteration on each sub-system of equations, in a global Gauss-Seidel iterative structure. Note that applying one Gauss-Seidel iteration for each X_i in the global Gauss-Seidel structure reduces the block Gauss-Seidel method to the standard Gauss-Seidel method, although the method is based on block sub-MVPs.

The test for convergence (Line 9 in Figure 3.1) is carried out using the relative error criterion given by Equation (2.16) on Page 18. This is accomplished as follows. In the i-th phase of the algorithm, using Equation (2.16), we compute the maximum relative error for the i-th block of the iteration vector. These computations are repeated for each phase (i.e. for each vector block) in order to calculate the maximum relative error for the whole iteration vector. This maximum relative error is finally compared to a preset

value of ϵ (in this thesis, we use $\epsilon = 10^{-6}$) before proceeding to the next iteration (Line 0 in Figure 3.1). Actually, during an iteration, the computations given by Equation (2.16) are only applied *until* the relative maximum error remains smaller than the preset value of ϵ.

3.2 The Compact MSR Storage Scheme

We have surveyed sparse matrix storage schemes in Chapter 2. In this section we describe the *compact Modified Sparse Row* (compact MSR) scheme, a matrix storage scheme which is a contribution of this thesis. We have used this scheme to store CTMC matrices in all our implementations of the solutions which are based on explicit matrix storage. The scheme can be viewed as a compact variation of the Modified Sparse Row (MSR) scheme, which has been described in Chapter 2. It offers significant savings over MSR and other sparse storage schemes. In the case of out-of-core solutions, these savings in disk memory as well as RAM are reflected in faster solution times due to better caching and reduced I/O.

It has also been mentioned in Chapter 2 that the indexed variations of the MSR scheme exploit the fact that the number of entries in a generator matrix is relatively small. We have also seen that the indexed MSR format (largely true for all mentioned formats) stores the column index of a nonzero entry in a matrix as a data type int. An int usually uses 32 bits, which can store a column index as large as 2^{32} (above 4 billion). The size of the models which can be fitted within the RAM of a modern workstation are much smaller

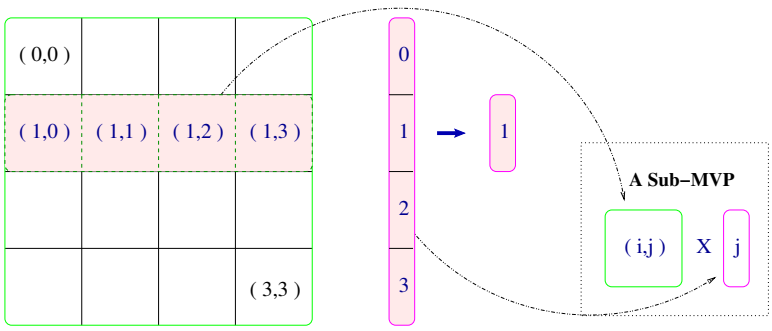


Figure 3.2: Matrix vector multiplication at block level

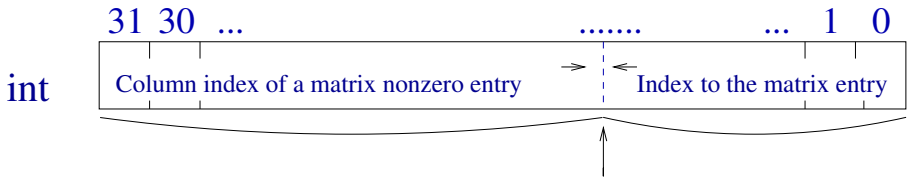


Figure 3.3: The idea of the compact MSR scheme

than 2^{32}. For example, it is evident from Table 2.1 on Page 22 that using an in-core method such as Gauss-Seidel, today, we can hardly solve a model with 54 million states on a 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 or row index inside each matrix block. In these cases, the upper bound on a row or column local index can be decreased by increasing the number of blocks or processes.

The compact MSR format 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 3.3. The storage and retrieval of these indices into, and from an int can be carried out efficiently using shift and mask 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 (see Section 2.6.1, Page 20). 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, therefore, uses three arrays: an array `Col_i` of length $4a$ bytes which stores the column positions of matrix entries as well as the indices to these entries, an n-byte long array `row_entries` to store the number of entries in each row, and a $2n$-byte long array `Diag_i` 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.

We compare now the storage requirements of the compact MSR scheme with other sparse schemes discussed earlier in Chapter 2. We focus for these

k	states (n)	a/n	memory required for Q (MB)			MB per π
			MSR format	*Indexed MSR*	*Compact MSR*	
FMS models						
8	4,459,455	8.64	489	225	160	34
9	11,058,190	8.96	1,260	577	409	84
10	25,397,658	9.23	2,962	1,366	967	194
11	54,682,992	9.47	6,554	3,016	2,133	417
12	111,414,940	9.68	13,563	6,279	4,433	850
13	216,427,680	9.87	26,923	12,429	8,768	1,651
14	403,259,040	12.35	61,609	28,882	20,152	3,077
15	724,284,864	12.61	112,810	52,952	36,913	5,526
Kanban models						
4	454,475	8.75	51	23	17	3
5	2,546,432	9.60	218	142	101	19
6	11,261,376	10.27	1,456	674	475	86
7	41,644,800	10.82	5,264	2,613	1,835	318
8	133,865,325	11.26	18,843	8,783	6,153	1,021
9	384,392,800	11.64	55,428	25,881	18,109	2,933
10	1,005,927,208	11.97	149,310	69,858	48,811	7,675

Table 3.1: Comparison of sparse storage formats

sparse schemes on the amount of RAM required during the steady-state, iterative solution phase. Table 3.1 reports these storage requirements in MSR, indexed MSR, and compact MSR formats for the flexible manufacturing system (FMS) of [CT93], and the Kanban manufacturing system of [CT96]. The first column in Table 3.1 gives the model parameter k; see Appendix B for details. Second column lists the resulting number of reachable states. Third column (a/n) lists the average number of nonzero entries per row.

The largest model reported in the table is of the Kanban system with over 1 billion reachable states and 12 billion transitions. Column 4 and 6 give the storage requirements for matrices (including storage for diagonal) in MB for MSR and compact MSR respectively. The amount of RAM required for the indexed MSR format, listed in column 5, excludes the storage for diagonal because it can be kept on disk and is not required during the iteration phase. Finally, the last column lists the memory required to store a single iteration vector of doubles (8 bytes) for the steady-state solution phase.

We note in Table 3.1, that the compact MSR scheme is around 30% more compact than the indexed MSR format. This significantly more compact storage reduces the RAM and (or) disk memory requirements of the (in-core or out-of-core) solution, and therefore, using the compact MSR scheme, it is possible to solve larger models on a particular hardware. Secondly, compact storage improves the speed of the solution due to reduced memory I/O and better caching. Furthermore, smaller amounts of data to move from disk mean faster solution due to reduced disk I/O.

3.3 Out-of-Core Algorithms

Programmers want to have access to an unlimited amount of fast memory. To satisfy these needs, most modern computers have a memory hierarchy organised into several levels, typically ranging from the fastest level of CPU registers to random access memory and (slower but inexpensive) magnetic disks. The goal is to provide a memory system with cost of the cheapest memory level and speed of the fastest.

Most modern computing systems provide the concept of a virtual memory, which enables programmers to access memory larger than the available RAM. A *virtual* memory system also means that programmers do not need to

worry about the actual location of the data in the memory hierarchy. System mechanisms, such as *caching*, *swapping* and *demand paging*, have been developed to control the data movement across the different levels of the memory hierarchy. The virtual memory systems, however, are usually designed to be general-purpose, and therefore, do not always fulfil programmers' expectations, in particular, when the ratio of virtual to physical memory sizes is large. Moreover, applications may still exist that require the amount of space much larger than the virtual memory can address. Distributed computing systems, such as clusters of workstations, provide a natural way of dealing with the memory (and computing) problems. However, as we have seen in Section 2.8.2, there is still a need to solve even larger problems, i.e. the problems for which the required memory can exceed the main memory available. In these cases, where the primary or virtual memory is unable to hold the application data, the so-called *out-of-core* methods can be used.

Out-of-core algorithms are those which are designed to attain high performance when their data structures are stored on disks. These algorithms can be classified as a sub-category of more general algorithms, known as *external memory* algorithms, which are designed to achieve performance by explicit and efficient control of the memory hierarchy. The two terms, out-of-core and external memory, have also been used interchangeably in the literature. We concentrate on the out-of-core algorithms, i.e. on the explicit control of the communication between the main memory (RAM) and disk, and, in this section, briefly explore the general issues involved in the design and analysis of such algorithms. For further details on out-of-core algorithms see e.g. [Vit01, Tol99]; see also [HP03], an excellent book on computer architecture.

The principle of *locality* plays a vital role in obtaining performance from algorithms. It states that programs tend to reuse data (and code) they have used recently. The concept of locality is usually divided into *temporal locality*, which states that recently accessed items are likely to be accessed in the near future, and *spatial locality*, which states that items whose addresses are near one another tend to be referenced close together in time [HP03].

An implication of this concept of locality, for out-of-core algorithms, is that the data must be laid out on disk such that it can be moved in large blocks between RAM and disk. Secondly, the computations must be performed such that the data which has been brought in is completely used

before its eviction from RAM. Moreover, the computations must be ordered such that the I/O between the two memory levels is minimised. For example, iterative methods which perform vector-matrix computations in a block-by-block manner, as we have seen in Section 3.1, provide high-level of data locality for out-of-core algorithms. The matrix and/or vector can be divided into blocks, and these blocks can be stored on disk in contiguous locations, hence exploiting spatial locality in the algorithm.

It is possible that a particular algorithm does not admit an efficient out-of-core scheduling. In this case, the original computations may be transformed in order to achieve performance from the out-of-core algorithm. For example, a block-based iterative algorithm can be considered as an efficient transformation of the corresponding (standard) iterative method, assuming that the standard method performs the matrix computations in a row or column wise fashion.

Creating locality may also bring some additional cost to the out-of-core algorithm. This cost may be incurred in that the resulting algorithm performs significantly larger amount of work than the original algorithm, degrading the overall out-of-core performance. Furthermore, although the out-of-core algorithm was kept equivalent to the original algorithm in exact arithmetic, it may become numerically unstable due to floating point arithmetic. Numerically stable out-of-core algorithms are usually preferred over less stable counterparts.

In addition to numerical stability, another property which is desirable in an out-of-core algorithm is its predictability in terms of the amount of space and time. One can ask questions, for example the following. How much RAM and disk memory the out-of-core algorithm requires? What is the impact of the amount of available RAM on the out-of-core performance of the algorithm? Are the time and space requirements of the algorithm predictable for larger sizes of the problem? How does the algorithm scale with the increase in the problem size? Is the amount of space and time linear in problem size? How large benchmarks have been used to demonstrate the performance of the algorithm? How does the benchmark sizes compare with the main memory of the workstation used? Is the workstation used, a fairly standard one, or a special computing equipment? What is the ratio of the largest problem sizes solved using in-core and out-of-core?

An out-of-core algorithm can also be analysed by comparing its performance against the in-core version of the algorithm. How do the memory requirements of the two versions compare? What is the ratio of the main memory requirements for the two versions? What is the ratio of the main memory for the in-core and the total disk space for the out-of-core? How does the solution time for the two versions compare? Does the out-of-core solution result in a many-fold increase in the in-core solution time? Is the increase in the solution time worth using the out-of-core algorithm?

A comparative analysis of an out-of-core algorithm can be made wherever one or more baseline algorithms are available. The amount of I/O performed by the individual algorithms can be compared. Their memory (both RAM and disk) requirements and solution times can be studied. The data layout in the main memory as well as on disk can be investigated, in order to compare the extent to which principal of locality has been exploited by the algorithms.

3.4 A Matrix Out-of-Core Algorithm

The block Gauss-Seidel algorithm given in Section 3.1 can be implemented for the steady-state solution of CTMCs such that the CTMC matrix can be stored using, for example, the compact MSR sparse scheme, and the vector can be stored using an array of length n doubles, where n is the number of states in the CTMC. However, this makes the total memory requirements for large CTMCs well above the size of the RAM available in standard workstations (see Table 3.1). One possible solution is to use an out-of-core approach, i.e. to store the matrix on disk and read blocks of matrix into RAM when required. In each iteration of the iterative method, we can do the following:

```
while there is a block to read
    read a matrix block
    use the matrix and vector blocks to compute the sub-MVP
    perform additional computations
```

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, in particular for large models and for iterative

methods, because they require a relatively large amount of data per floating point operation. To overcome these inefficiencies, we would like to use a two-process approach where the disk I/O and the computation can proceed concurrently. We present such an approach in this section. Once we have explained the working of a basic out-of-core iterative algorithm, we will be able to consider out-of-core algorithms which are more involved. Furthermore, it will allow us to make a comparison of various out-of-core approaches pursued in the literature.

Figure 3.4 presents a high-level description of a matrix out-of-core algorithm which uses the block Gauss-Seidel method for the solution of the system $Ax = 0$. The name "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 3.4 assumes that the CTMC matrix to be solved is divided into P^2 blocks of size $n/P \times n/P$, and is stored on disk. This is called a uniform checkerboard partitioning of the matrix. In fact, for this algorithm, it is not necessary to divide the matrix into P^2 blocks. It suffices to have P matrix blocks of size $n/P \times n$, i.e. P matrix blocks, where each matrix block consists of n/P rows of the CTMC matrix. This corresponds to the rowwise block-striped partitioning of the CTMC matrix; see [KGGK94] for details of matrix partitioning, for instance.

For clarity reasons, in Figure 3.4, the vector x is also shown to be divided into P blocks, although a single array of doubles is used to keep the whole vector. Another vector $\tilde{X}_i$ of size n/P is required to accumulate the sub-MVPs (line 8). Only one $\tilde{X}_i$ vector is used, the subscript i is used for clarity reasons. The algorithm assumes that, before it commences, the array x holds an initial approximation to the solution.

As for the in-core Gauss-Seidel algorithm of section 3.1, each iteration of the out-of-core algorithm given in Figure 3.4 progresses in P 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 $0 \leq j < P$; see Figure 3.2. The *DiskIO Process* helps the *Compute Process* with this task

Integer constant: P (*number of blocks*)
Semaphores: S_1, $S_2 \leftarrow$ `occupied`
Shared variables: R_0, R_1 (*To read row of matrix blocks into RAM*)

DiskIO Process

1. Local variables: i, j, $t = 0$
2. `while` not converged
3. `for` $i \leftarrow 0$ to $P - 1$
4. read all nonzero A_{ij}
5. `Signal`(S_1) /* over R_t */
6. `Wait`(S_2) /* over $R_{\bar{t}}$ */
7. $t = \bar{t}$
8. `end for`
9. `end while`

Compute Process

1. Local variables: i, j, $t = 0$
2. `while` not converged
3. `for` $i \leftarrow 0$ to $P - 1$
4. $\tilde{X}_i \leftarrow 0$
5. `Wait`(S_1) /* over R_t */
6. `Signal`(S_2) /* over $R_{\bar{t}}$ */
7. `for` all nonzero $A_{ij} \mid j \neq i$
8. $\tilde{X}_i = \tilde{X}_i - A_{ij}X_j$
9. `end for`
10. Perform $A_{ii}X_i = \tilde{X}_i$
11. Test for convergence
12. $t = \bar{t}$
13. `end for`
14. `end while`

Figure 3.4: A matrix out-of-core block Gauss-Seidel algorithm

and fetches the required row of the blocks of matrix A from disk (line 4). Note that in the *DiskIO Process*, the whole row of matrix blocks is fetched at once in a single buffer. Once the i-th row of blocks is in-core, the *Compute Process* can perform the numerical computations given by lines $8, 10$ and 11. Line 8 in the process accumulates the sub-MVPS, and line 10 solves the equation for X_i, using a Gauss-Seidel (inner) iteration. Line 11 performs a test of convergence in part using the previous and newly calculated entries of the vector block X_i. These computational steps have been explained in detail in Section 3.1.

The algorithm in Figure 3.4 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 row of matrix blocks in one shared buffer, say R_0, while the *Compute Process* is consuming a row of matrix blocks from the other buffer, R_1. Both processes alternate the value of a local boolean variable t, 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 over these buffers; see, for instance, lines $5 - 6$ in the *DiskIO Process*.

The high-level structure of the algorithm, given in Figure 3.4, is that of a producer-consumer problem. In each execution of its `for` loop, the i-th phase (lines $3 - 8$), 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* waits on S_1 (line 5). On receiving this signal, the *Compute Process* issues a return signal on S_2 (line 6) and then advances to update the i-th vector block (lines $7 - 10$). 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 read or updated; the two processes then advance asynchronously to the next iteration.

3.4.1 Implementation

We have used the compact MSR storage scheme (see Section 3.2) 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 and no file seek operations are required during an iteration.

We have implemented the algorithm using shared memory and semaphores, the Inter Process Communication (IPC) mechanisms available on Unix-like operating systems. Another possibility is to use a single process comprising two threads. In this case, the two threads will share a single address space and, depending on the underlying operating system, this approach may result in faster solution speed.

3.5 The Complete Out-of-Core Algorithm

A limitation of the solution method based on a matrix out-of-core approach is the in-core storage of the iteration vector. In this section, we introduce our complete out-of-core algorithm which stores the CTMC matrix as well as the iteration vector on disk. As for the matrix out-of-core algorithm, the complete out-of-core algorithm employs the block Gauss-Seidel algorithm of Section 3.1 to solve the system $Ax = 0$.

The high-level description of the complete out-of-core algorithm for the block Gauss-Seidel solution of the system $Ax = 0$ is shown in Figure 3.5. The algorithm assumes that the iteration vector and the CTMC matrix are divided into P and P^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_{P-1}, is already in-core.

As for the matrix out-of-core, the complete out-of-core algorithm of Figure 3.5 uses two shared memory buffers, R_0 and R_1, to read blocks of matrix into RAM from disk (line 8 of the *DiskIO Process*). Similarly, vector blocks are read (line 10) 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 11, *Compute Process*). The subscript i for $\tilde{X}_i$, as mentioned earlier, is only for clarity reasons. The local boolean variable t is used to switch between the pair of shared buffers, R_t and $Xbox_t$. We note in the figure that, in the i-th phase of an iteration, the two processes wait over different sets of shared memory buffers, i.e. the *DiskIO Process* (line 14) waits

Integer constant: P (*number of blocks)*
Semaphores: S_1, $S_2 \leftarrow$ `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, t = 0
2. k ← P − 1
3. while not converged
4.   for i ← 0 to P − 1
5.     j ← i ? (i − 1) : (P − 1)
6.     for h ← 0 to P − 1
7.       if not an empty block
8.         read A_ij from disk
9.         if h ≠ 0
10.          read X_j from disk
11.        end if
12.      end if
13.      Signal(S_1)   /* R_t, Xbox_t */
14.      Wait(S_2)   /* R_t̄, Xbox_t̄ */
15.      if h = 0
16.        write X_k to disk
17.      end if
18.      j ← j ? (j − 1) : (P − 1)
19.      t = t̄
20.    end for
21.    k ← i
22.  end for
23. end while
```

Compute Process

```
1. Local variable: h, i, j, t = 0
2. while not converged
3.   for i ← 0 to P − 1
4.     j ← i ? (i − 1) : (P − 1)
5.     X̃_i ← 0
6.     for h ← 0 to P − 1
7.       Wait(S_1)   /* R_t, Xbox_t */
8.       Signal(S_2)   /* R_t̄, Xbox_t̄ */
9.       if j ≠ i
10.        if not an empty block
11.          X̃_i ← X̃_i − A_ij X_j
12.        end if
13.      end if
14.      else
15.        Perform A_ii X_i = X̃_i
16.        Test for convergence
17.      end else
18.      j ← j ? (j − 1) : (P − 1)
19.      t = t̄
20.    end for
21.  end for
22. end while
```

Figure 3.5: The complete out-of-core block Gauss-Seidel iterative algorithm

over the shared buffers $R_{\bar{t}}$ and $Xbox_{\bar{t}}$, while the *Compute Process* (line 7) waits over the buffers R_t and $Xbox_t$ (shown as comments in the algorithm). A similar explanation applies to the signal operations for the two processes in the algorithm. For the sake of algorithm layout and simplicity, we have abstracted some information from the algorithm regarding the shared memory buffers.

The vector blocks corresponding to empty matrix blocks are not read from disk (lines 7, 10, *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; this information is abstracted from the algorithm. Once a vector block has been updated, this new approximation of the block must be updated on disk (lines 15, 16, *DiskIO Process*). The variable k (lines 2, 21, *DiskIO Process*) is used to keep track of the index of the vector block to be written during an inner iteration (lines $6-20$, *DiskIO Process*). The variable j (lines 5 and 18, *DiskIO Process*) is used to keep track of the matrix and vector blocks to be read from disk, and to be consumed by the Compute Process (lines 4 and 18). The variable j for the two processes is updated using a conditional statement; line 4 in the *Compute Process*, for instance, is explained as:

$$\texttt{if}(i = 0) \ \texttt{then}\ j \leftarrow P - 1 \ \texttt{else}\ j \leftarrow i - 1.$$

Line 18 in the *Compute Process* can be explained similarly. We use this notation to improve the readability of the algorithm. It is a standard notation used for conditional statements in the C programming language. The *Compute Process* accumulates the sub-MVPs at line 11, and solves the subsystems of equations for X_i (line 15) using a Gauss-Seidel (inner) iteration, as explained in Section 3.1.

Finally, we note that, in the algorithm, the diagonal blocks for vector and matrix ($j = i$) will always be consumed (and read) when $h = P - 1$, i.e. in the last iteration of the inner `for` loop; see lines $14 - 17$ in the *Compute Process*. In the light of this discussion, and the explanation of the algorithms given in Sections 3.1 and 3.4, the *Compute Process* is self explanatory.

3.5.1 Implementation

We have used two separate files to store the matrix and the iteration vector on disk. The matrix is stored in the compact MSR scheme and the vector is stored as an array of doubles. As for the matrix out-of-core solution, the matrix for the complete out-of-core method has been stored on disk, block by block, 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. Another array of size P^2 can be allocated in the main memory to keep track of the zero and nonzero matrix blocks. Actually, we have used a sparse scheme to store this information regarding nonzero blocks. The number of blocks P for the complete out-of-core solution is small, usually less than 100, and therefore the memory required for the array is negligible.

The complete out-of-core method was originally introduced in [KM02]. There, we described another implementation of the complete out-of-core algorithm which exploits the sparsity pattern in the CTMC matrices. Further research can be carried out to develop intelligent out-of-core algorithms which are able to dynamically exploit sparsity patterns available in a CTMC matrix.

3.6 Experimental Results

We now analyse the performance of the algorithms which have been presented in this chapter. We have implemented these algorithms on an UltraSPARC-II 440MHz CPU machine running SunOS 5.8 with 512MB RAM, and a 6GB local disk. We have used three widely used benchmark models to test the implementations. These are: a flexible manufacturing system (FMS) of [CT93], a Kanban system of [CT96] and a cyclic server Polling system of [IT90]. We will use these case studies to benchmark all our solution techniques, throughout this thesis. The models were generated using version 1.3.1 of the PRISM tool; see Appendix A.

This results section is organised as follows. In the following subsection, we say a few words on the file generation process for the out-of-core solutions. Subsequently, in Section 3.6.2, we present and compare times per iteration

for in-core and out-of-core iterative solutions. Finally, the matrix and the complete out-of-core solutions are individually analysed in Section 3.6.3 and Section 3.6.4, respectively, with the help of performance graphs.

3.6.1 File Generation

The PRISM tool stores CTMC matrices using an MTBDD-based data structure. Since the out-of-core solutions require the matrix to be kept in the compact MSR format, the MTBDD-based representation of the CTMC matrix is first converted into sparse format, and is then written to disk. Once the matrix and the iteration vector are ready on disk, the out-of-core iterative steady-state solution process can be invoked.

The times to generate the files for the out-of-core solution phase are proportional to the times required to convert a model from MTBDD representation to a sparse format. This file generation process can be either optimised 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 file repeatedly as the conversion progresses. This memory optimisation enables us to generate very large models on a workstation with limited RAM. Given that sufficient RAM is available, 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. Essentially, the time to generate files is negligible compared to the numerical solution times. We hereon concentrate on the iterative solution phase.

3.6.2 A Comparison of the Solution Methods

We outline the experimental results in Table 3.2 for the Kanban, FMS and Polling system case studies. Column 2 gives the model parameter k, see Appendix B for details. The resulting number of reachable states, n, is 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 3.2 list the time per iteration results for the implementations of the algorithms presented in this Chapter. These include the standard in-core version, where the matrix and the vector are kept in RAM; the matrix out-of-core version (Section 3.4), where only the matrix is stored

Model	k	States (n)	a/n	Time (seconds per iteration)			Iter.
				In-core	Out-of-core		
					Matrix	Complete	
FMS	6	537,768	7.8	0.3	0.5	1.1	812
	7	1,639,440	8.3	1.1	1.7	3.8	966
	8	4,459,455	8.6	3.2	5.1	10.7	1,125
	9	11,058,190	8.9	–	24	51.8	1,287
	10	25,397,658	9.2	–	69	146	1,454
	11	54,682,992	9.5	–	–	374	1,624
Kanban	4	454,475	8.8	0.3	0.5	1.0	323
system	5	2,546,432	9.6	1.8	3.0	6.0	461
	6	11,261,376	10.3	–	30	68.6	622
	7	41,644,800	10.8	–	180	283	802
Polling	15	737,280	8.3	0.5	0.7	1.2	32
system	16	1,572,864	8.8	1.1	1.9	2.9	33
	17	3,342,336	9.3	2.4	3.9	6.4	34
	18	7,077,888	9.8	5.5	15.8	20.4	34
	19	14,942,208	10.3	–	41	71	35
	20	31,457,280	10.8	–	101	162	36
	21	66,060,288	11.3	–	–	359	36

Table 3.2: Comparing speeds for explicit in-core and out-of-core methods

on disk and the vector is kept in RAM; and the complete out-of-core version (Section 3.5), where both the matrix and the vector are stored on disk. Note that all these methods are based on explicit storage of CTMC matrix. We know from earlier sections in this chapter that the iterative method used for the numerical solution in all algorithms is the block Gauss-Seidel method. The number of blocks for these iterative solutions were manually selected, based on our analyses of the two out-of-core methods. These analyses will be presented in Sections 3.6.3 and 3.6.4. The number of blocks for each CTMC generator matrix is selected such that the resulting amount of memory required for its solution remains well within the available physical memory of the workstation. For a priori selection of the number of blocks, we discuss a heuristic in Section 3.6.5.

The number of iterations for each solution are reported in column 8, in Table 3.2. The criterion we have used to test for convergence, in all our implementations, is given by Equation (2.16) for $\epsilon = 10^{-6}$. All reported run times are *wall clock* times.

The CTMCs for all three solution methods are stored in the compact MSR sparse storage scheme (Section 3.2). The scheme requires $4a + 3n$ bytes to store the whole matrix including the diagonal. The entries "a/n" in column 4 can be used to calculate the memory required to store the matrices.

We note, in Table 3.2, 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 ($k = 18$) with approximately 7 million states. The upper bound on the size of the solvable model, in this case, is determined by the total amount of memory required for the storage of the matrix and the iteration vector. The matrix out-of-core solution requires in-core storage for one iteration vector and two blocks of matrix. The memory required for these matrix blocks can, in general, be reduced by increasing the number of blocks the matrix is decomposed into. However, in this case, the limit on the largest solvable model still exists due to the in-core storage of the iteration vector. Pursuing the matrix out-of-core approach, the largest model solvable on the workstation is the Kanban system ($k = 7$) with approximately 41 million states.

The upper bound on the size of solvable models can be increased by the out-of-core storage of both matrix and the iteration vector. This is evident

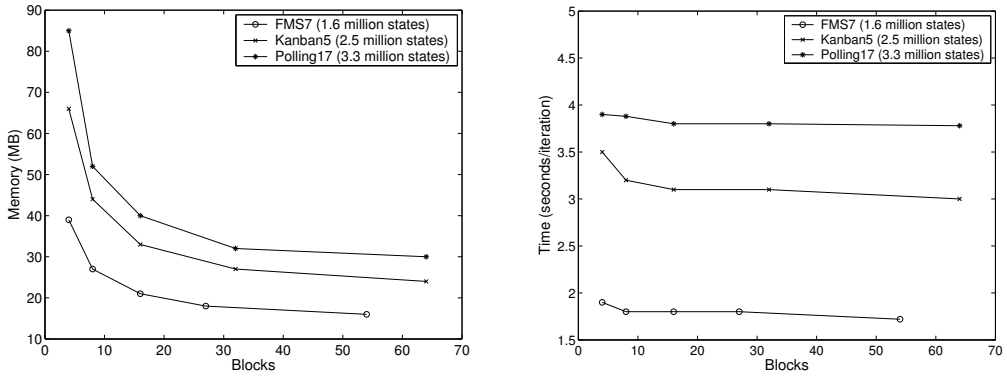
from column 7, and the largest model reported in this case is the Polling system ($k = 21$) with 66 million states. The limit in this case is the size of the available disk (6GB), although, using this approach, even larger models can be solved provided a larger disk is available.

3.6.3 The Matrix Out-of-Core Method

In this section, we investigate the performance for the matrix out-of-core solution method by analysing its memory and time properties plotted against the number of blocks used to partition the vector. A matrix out-of-core algorithm was given in Section 3.4 and the time per iteration results presented in Table 3.2. In Figure 3.6(a), we plot 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 P of equal size. This corresponds to P^2 matrix blocks of equal size.

We explain Figure 3.6(a) using the plot for the Polling system ($k = 17$). The memory required to store the entire vector and matrix in this case, if kept completely in RAM, is approximately 26MB and 135MB 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 $P = 4$ is 85MB. Increasing the number of blocks up to $P = 64$ reduces the memory requirements to nearly 30MB. This minimum is bounded by the storage required for the iteration vector (26MB). Similar properties are observed in the plots for the FMS ($k = 7$) and Kanban system ($k = 5$) CTMCs.

In Figure 3.6(b), we analyse the time per iteration characteristics for the same three CTMCs, plotted against the number of vector blocks. We note, in the figure, 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 (better caching and lesser memory I/O), as demonstrated in Figure 3.6(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 matrix blocks, which possibly have less variations in their sizes (number of nonzero entries), 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 (not shown in the



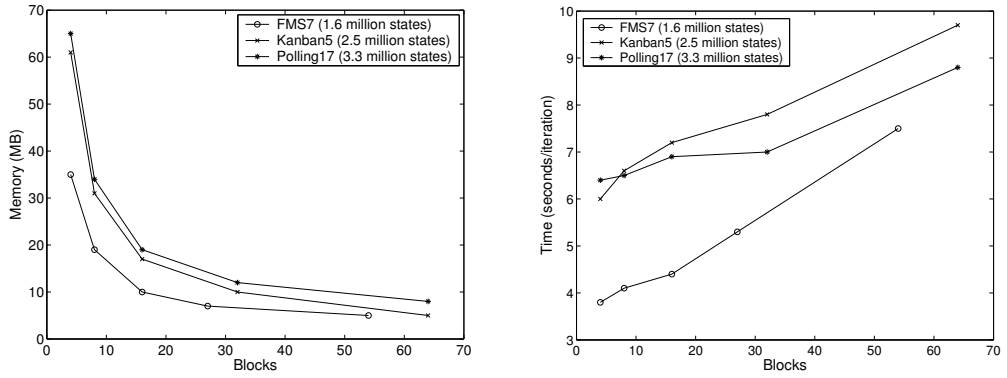
(a) Memory vs. blocks (b) Time per iteration vs. blocks

Figure 3.6: The matrix out-of-core method: space and time versus P

figure) will typically result in an increase in solution time due to a higher amount of memory which is required to store the block information. The higher number of synchronisation points in this case may also be a partial cause for the increase in solution times (see the algorithm in Section 3.4). Note that, although the matrix blocks have an equal number of rows, the variations in the sizes of the blocks is caused by varying and unequal number of nonzero entries.

3.6.4 The Complete Out-of-Core Method

We investigate here the performance for the complete out-of-core solution by analysing the memory and time properties plotted against the number of blocks. We have presented the complete out-of-core algorithm in Section 3.5 and have given the time per iteration results from its implementation in Table 3.2. Figure 3.7(a) illustrates the total memory requirement of the complete out-of-core solution against the number of blocks for the same three CTMCs. As for the matrix out-of-core method, the vector and the CTMCs are partitioned into P and P^2 blocks respectively, and, in this case, both are stored on disk. The memory plots in the figure show a similar trend as for the matrix out-of-core method illustrated in Figure 3.6(a). However, for the complete out-of-core solution, the minimum amount of memory is not bounded by the storage of the iteration vector. For example, for the Polling



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

Figure 3.7: The complete out-of-core method: space and time versus P

system ($k = 17$), the memory required for the case $P = 4$ is 65MB. Increasing the number of blocks up to $P = 64$ reduces the memory requirements to nearly 12MB.

The time per iteration properties of the complete out-of-core solution are plotted in Figure 3.7(b). The properties are in contrast with the matrix out-of-core method. With an increase in the number of blocks, we observe an increase in the solution times for this method. We explain the time properties of the method by considering the plot for FMS system ($k = 7$), which exhibits the highest rate of increase in time. The time per iteration in this plot increases from approximately a minimum of 4 seconds to a maximum of 7 seconds, an approximate increase ratio of 1.7. This increase in time is explained as follows. In the complete out-of-core solution, the whole iteration vector must be read from disk in order to compute the next approximation for each block of iteration vector. This implies that the iteration vector must be read P times in each iteration of the method. Consequently, an increase in the number of blocks typically results in an increase in the amount of disk I/O, and hence an increase in the solution time. A note, however, on the positive aspects of this increase in time: an approximate 14-fold increase in the number of blocks results in less than a 2-fold increase in solution time and a 7-fold decrease in the amount of memory. We observe similar patterns in the plots for the Polling and Kanban system CTMCs.

3.6.5 A Priori Selection of P

We conclude this section with our observation that the performance of the out-of-core solutions is determined by the number of blocks, P, that the matrix and vector are partitioned into. The time per iteration for the matrix out-of-core method remains unchanged (actually it decreases slightly) with an increase in the number of blocks. For the complete out-of-core method, the time per iteration increases with an increase in the number of blocks. The total memory required for both the methods decreases as the number of blocks increases. It follows from Figure 3.6 and Figure 3.7 that the two methods deliver a satisfactory performance (i.e., low memory and time per iteration) for a wide range of values of the number P. A priori estimation of a value for the number P which can yield a desirable performance, is hence possible. This is explained as follows. A function can be formulated for each out-of-core method. Given a number (of blocks) P, the function yields the amount of required RAM for the solution process. Given this function, a number P can be searched for such that the RAM associated to this number P remains under some predefined value (or under the available RAM in the workstation). Such a heuristic provides, for both methods, a nearly optimum solution time for a given amount of allocated RAM.

Further experimental analysis of the two out-of-core methods has revealed that the memory and time plots, as given in Figure 3.6 and Figure 3.7, demonstrate similar patterns for different values of k for each case study. This can, in fact, be useful for predicting good choices of P for larger values of k.

3.7 Discussion

A matrix out-of-core technique for the steady-state solution of Markov models was first considered by Deavours and Sanders [DS97] in 1997. The authors used a block Gauss-Seidel method in their tool to reduce the amount of disk I/O; see Section 2.3 for a description of block iterative methods in general, and Section 3.1 for a block Gauss-Seidel algorithm. Their idea was to partition the matrix into a number of sub-systems (or blocks) and apply multiple Gauss-Seidel *inner* iterations on each matrix block, i.e. to use the data multiple times. The number of inner iterations was a tunable parameter of their

disk-based tool. They analysed the tool by presenting results for a fixed and varying number of inner iterations. Later, in [DS98a], the authors improved their earlier work by applying a compression technique to the matrix before storing it to disk, hence reducing the file I/O time. Deavours and Sanders reported the solution of models with up to 15 million states on a workstation with 128MB RAM.

For our implementations of the out-of-core methods we have used the compact MSR scheme for matrix storage, which can also be considered as a replacement of their compression technique. The compact MSR scheme provides a 30% saving over conventional schemes, and, therefore, we believe that further file compression would not make much difference. Furthermore, Deavours and Sanders report that the decompression time accounts for 50% of the computation time. The compact MSR scheme, on the other hand, does not have a decompression cost.

The other notable papers concerning the out-of-core solutions based on explicit CTMC storage are [KH99] and [BH01], as mentioned in Section 2.8. The main emphasis of these papers is on the parallelisation of the matrix out-of-core approach of Deavours and Sanders. Obviously, the limitation of all these methods, be it serial or parallel, has been the memory (RAM) required to store the iteration vector(s). Our complete out-of-core approach coupled with the compact MSR data structure relaxes these limitations.

CHAPTER

FOUR

A Symbolic Out-of-Core Solution Method

The previous chapter presented a study of out-of-core iterative methods for the steady-state solution of CTMCs, based on explicit storage of the CTMC matrix. We now turn our attention to steady-state solution techniques which are based on implicit CTMC storage. In this context, we use offset-labelled MTBDDs to implement and analyse a novel symbolic out-of-core solution method.

We organise this chapter as follows. Firstly, we give the motivation for the symbolic out-of-core technique in Section 4.1. The Pseudo Gauss-Seidel method, the iterative method we have employed with the symbolic out-of-core algorithm is described in Section 4.2. The symbolic out-of-core algorithm itself is presented in Section 4.3. We experiment with two different implementations of the symbolic out-of-core algorithm, and describe these in Section 4.4 and Section 4.5. The difference between the two implementations lies in the way the CTMC matrix is partitioned into blocks. Experimental results for the symbolic solution method are presented in Section 4.6. The two implementations are compared in Section 4.6.1, and further explored in Section 4.6.2. In Section 4.6.3, we present a heuristic for a priori selection of the number of vector blocks P. Finally, in Section 4.6.4, we compare the symbolic and explicit methods (see previous chapter, Section 3.6.2).

4.1 Motivation

Recall from Chapter 2 that the computation of the steady-state probabilities of a CTMC can be reduced to solving a linear equation system and well-known iterative methods can be used for the solution. Also recall from Section 3.1 that the Gauss-Seidel iterative method for the solution of systems of linear equations of the form $Ax = 0$ requires storage for the CTMC matrix and for the iteration vector. This makes the total memory requirement for the solution of large CTMCs well above the size of the RAM available in standard workstations. Consequently, we have seen the matrix out-of-core approach of Section 3.4 (originally from Deavours and Sanders [DS97, DS98a]), where the matrix can be stored on disk and the blocks of matrix can be read into RAM when required. The in-core storage of an iteration vector in this case can still be prohibitive. We then learnt in Section 3.5 that our complete out-of-core approach can resolve the vector storage problem by keeping the matrix and vector both on disk. The out-of-core scheduling of both the matrix and the iteration vector, however, incurs a penalty in terms of disk I/O. We now explore another approach. We keep the CTMC matrix in-core, in a compact, for example, implicit representation, and store only the iteration vector on disk. This can greatly reduce the disk I/O penalty while at the same time allowing for larger models to be analysed. This motivates our symbolic out-of-core solution described in this chapter.

The idea of the symbolic out-of-core method is to keep the matrix in-core, in an appropriate symbolic data structure, and to store the probability vector on disk. The iteration vector is divided into a number of blocks. During the iterative computation phase, these blocks can be fetched from disk, one after another, into main memory to perform the numerical computation. We have used offset-labelled MTBDDs for CTMC storage, while the iteration vector for numerical computation is kept on disk as an array. Note that it is equally possible to use other implicit methods such as those based on Matrix Diagrams [CM99] or Kronecker expressions [Pla85], as long as the data structure holding the matrix allows an efficient decomposition of the CTMC matrix into its submatrices.

It is interesting to note here the distinctions among the three out-of-core approaches. In the matrix out-of-core approach, the matrix alone is kept on disk, while the symbolic out-of-core method stores only the iteration vector

on disk; both the vector and matrix are stored on disk for the complete out-of-core method.

4.2 The Pseudo Gauss-Seidel Method

In this section, we describe the pseudo Gauss-Seidel iterative algorithm for the solution of the linear equation system $Ax = 0$. We are already aware of the motivation for this iterative method from Chapter 2. It is reiterated here as follows.

Implementations of the steady-state solution for CTMCs which are based on the Gauss-Seidel iterative method are attractive for two reasons. Firstly, the method converges faster than its counterpart Jacobi. Secondly, its implementation requires storage of a single iteration vector while Jacobi method requires two. Unfortunately, the offset-labelled MTBDD data structure does not allow an efficient implementation of Gauss-Seidel. The Gauss-Seidel method requires row-wise access to matrix entries; however, a depth-first traversal of the MTBDD data structure does not allow matrix entries to be extracted in this order. This problem is resolved by the the pseudo Gauss-Seidel method [Par02], a compromise between Jacobi and Gauss-Seidel. The method is summarised as follow.

We concentrate on the solution of the system $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. We use here the partitioning of a matrix as given in Section 2.3, with some additional notations required to explain the pseudo Gauss-Seidel method. Assume that the state space S of the CTMC is divided into P contiguous partitions $S_0, \ldots, S_{P-1}$ of sizes $n_0, \ldots, n_{P-1}$, such that $n = \sum_{i=0}^{P-1} n_i$. No assumptions are made as regards to the relative sizes of these partitions. Using this, the matrix A can be divided into P^2 blocks, $\{A_{pq} \mid 0 \leq p, q < P\}$, where the rows and columns of block A_{pq} correspond to the states in S_p and S_q, respectively, i.e. block A_{pq} is of size $n_p \times n_q$. We introduce the additional notation $N_p = \sum_{i=0}^{p-1} n_i$, for $0 \leq p \leq P$. A partition S_p includes states with indices N_p up to $N_{p+1} - 1$. We also define $n_{\max} = \max\{n_p \mid 0 \leq p < P\}$. Finally, we denote by $block(i)$ the index of the block containing state i, i.e. the unique $0 \leq p < P$ such that $N_p \leq i < N_{p+1}$. The k-th iteration of the pseudo Gauss-Seidel method comprises the computation:

1. `while` not converged
2. `for` $p = 0$ to $P - 1$
3. $\tilde{X}_p \leftarrow 0$
4. `for` $q = 0$ to $P - 1$
5. $\tilde{X}_p \leftarrow \tilde{X}_p - \check{A}_{pq} X_q$
6. `end for`
7. $\tilde{X}_p \leftarrow D_{pp}^{-1} \tilde{X}_p$
8. Test for convergence
9. $X_p \leftarrow \tilde{X}_p$
10. `end for`
11. Stop if converged
12. `end while`

Figure 4.1: The pseudo Gauss-Seidel algorithm

$$x_i^{(k)} = -\frac{1}{a_{ii}} \left(\sum_{j<N_{block(i)}} a_{ij}\ x_j^{(k)} + \sum_{j \geq N_{block(i)} \wedge j \neq i} a_{ij}\ x_j^{(k-1)} \right) \tag{4.1}$$

for $0 \leq i < n$. We additionally introduce the matrices D and $\check{A}$, which contain the diagonal and off-diagonal elements of A respectively, and write the above equation in a block notation form:

$$X_p^{(k)} = -D_{pp}^{-1} \left(\sum_{q<p} \check{A}_{pq}\ X_q^{(k)} + \sum_{q \geq p} \check{A}_{pq}\ X_q^{(k-1)} \right) \tag{4.2}$$

for $0 \leq p < P$, where $X_p^{(k)}$ and $X_p^{(k-1)}$ are the p-th blocks of vectors $x^{(k)}$ and $x^{(k-1)}$ respectively. As above, F_{pq} denotes the (p, q)-th block of a matrix F. Compare the above equation with the block Gauss-Seidel equation (3.1).

The pseudo Gauss-Seidel method can be formulated into an MVP-based algorithm, as shown in Figure 4.1. The algorithm works as follows. Each iteration progresses in P phases. In the p-th phase, the method updates elements in the p-th block of the solution vector. It does this using the most recent approximation for each element of the solution vector available, i.e.

it uses values from the previous iteration for entries corresponding to vector blocks $p, \ldots, P-1$ and values from earlier phases of the current iteration for entries corresponding to blocks $0, \ldots, p-1$; see Equation (4.2). The pseudo Gauss-Seidel method can be related to the Jacobi and Gauss-Seidel methods by considering Jacobi to be the case where $P = 1$ and Gauss-Seidel to be the case where $P = n$.

Note in Figure 4.1, that the p-th phase of an iteration, which computes the p-th block of the solution vector, only requires access to entries from the p-th row of blocks in $\breve{A}$, i.e. $\breve{A}_{pq}$ for $0 \leq q < P$. This is the same as for the block Gauss-Seidel algorithm of Section 3.1. We have illustrated this in Chapter 3, Figure 3.2, for $p = 1$ and $P = 4$. All blocks in the figure are of equal size but, as for the block Gauss-Seidel algorithm of Section 3.1, this is generally not the case. A unit of computation in the algorithm comprises the multiplication of a single matrix block by a single vector block, i.e. a sub-MVP. This corresponds to line 5 of the algorithm in the figure.

The algorithm given in Figure 4.1 requires, in addition to the matrix storage, one iteration vector x of size n to store the solution vector, the p-th block of which is denoted X_p, and another vector $\tilde{X}_p$ of size $n_{\max}$ to accumulate the sub-MVPs. Only one array $\tilde{X}_p$ is used; the subscript p of $\tilde{X}_p$ in the algorithm is used to make the description intuitive and to keep the vector block notation consistent. The reader may find it interesting to compare the pseudo Gauss-Seidel algorithm with the block Gauss-Seidel algorithm (Section 3.1).

The pseudo Gauss-Seidel iterative method uses some elements of the most recent approximation in each iteration, and therefore, it typically converges faster than the Jacobi method. Factors which affect the speed of its convergence include the number of partitions, P, and the sizes of these partitions. It can be shown, that the convergence characteristics of pseudo Gauss-Seidel are similar to those of Jacobi and Gauss-Seidel, as presented for example in [Ste94]; see Parker [Par02] for the details.

4.3 The Symbolic Out-of-Core Algorithm

In this section, we describe a symbolic out-of-core algorithm for the iterative solution of the linear equation system $Ax = 0$. The iterative method we

use to solve the system $Ax = 0$, the pseudo Gauss-Seidel method, has been explained in the previous section.

In Figure 4.2, we present a high-level description of the symbolic out-of-core algorithm for the numerical solution of the linear system $Ax = 0$. As for the explicit out-of-core methods considered in Chapter 3, the symbolic algorithm consists of two concurrent processes, the *DiskIO Process* and the *Compute Process*. The two processes communicate via shared memory and synchronise with semaphores.

The algorithm of Figure 4.2 assumes that the CTMC matrix to be solved is stored in-core, using the offset-labelled MTBDD data structure. We use the partitioning of a CTMC given in Section 4.2. The CTMC to be analysed is divided into P^2 partitions of sizes $n_0, \ldots, n_{P-1}$, such that $n = \sum_{i=0}^{P-1} n_i$. The largest partition is defined as $n_{\max} = \max\{n_p \mid 0 \leq p < P\}$. To preserve structure in the symbolic representation of matrix A, the diagonal entries of the matrix A are stored separately. The matrix $\check{A}$ and the vector d contain the off-diagonal and the diagonal entries of the matrix A, respectively. Therefore, using the above partitioning of the CTMC, the off-diagonal matrix $\check{A}$ is divided into P^2 blocks, $\{\check{A}_{pq} \mid 0 \leq p, q < P\}$, i.e. block $\check{A}_{pq}$ is of size $n_p \times n_q$. An illustration of the decomposition of a matrix into 4^2 blocks of equal sizes can be seen in Figure 3.2 on Page 42.

These off-diagonal matrix blocks are kept together as one offset-labelled MTBDD. Pointers to the MTBDD nodes representing each block are stored in a data structure to allow fast access during the sub-MVP operation (line 8, *Compute Process*). We have experimented with two implementations of the algorithm given in Figure 4.2. In the first implementation, we have divided the matrix into blocks of equal sizes, i.e., blocks with equal number of rows and columns. The second implementation uses the natural partitioning of the MTBDD data structure which yields blocks of different sizes. We consider these implementation details in the next two sections, and concentrate here on the high-level description of the symbolic out-of-core algorithm.

The probability vector x is also divided into P blocks (using the same partitioning as the matrix) of sizes $\{n_0, n_1, \ldots n_{P-1}\}$, where the p-th block is denoted by X_p. Before the algorithm commences, it assumes that an initial approximation for the probability vector x has been stored on disk and that the block X_{P-1} is already available in RAM. In order to schedule the vector

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

DiskIO Process

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

Compute Process

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

Figure 4.2: The symbolic out-of-core pseudo Gauss-Seidel algorithm

x out-of-core, the algorithm requires three arrays of size $n_{\max}$ doubles. The array $\tilde{X}_p$, which is local to the *Compute Process*, is used to accumulate the sub-MVPs (line 8, *Compute Process*). As earlier, the subscript p for $\tilde{X}_p$ is for intuitive reasons. The other two arrays required are the shared memory buffers, $Xbox_0$ and $Xbox_1$. These are used to read vector blocks from disk (line 8, *DiskIO Process*). At a certain point in time during the execution, the *DiskIO Process* is reading a block of iteration vector in one shared buffer, say $Xbox_0$, while the *Compute Process* is consuming a vector block from the other buffer, $Xbox_1$, to accumulate the sub-MVP $\check{A}_{pq}X_q$. Both processes alternate the value of a local boolean variable t, in order to switch between the two buffers $Xbox_0$ and $Xbox_1$.

As mentioned earlier, the diagonal elements of the CTMC matrix are stored separately as a vector d. Since the number of the distinct values in the diagonal of the matrices considered in our experiments is relatively small, n short int indices to an array of these distinct values are stored instead of n doubles. Also, to save n divisions per iteration, the value val of each distinct diagonal entry is stored as $1/val$. The diagonal vector is divided into P blocks of sizes $\{n_0, n_1, \ldots n_{P-1}\}$, in conformity to the block sizes of the matrix and the iteration vector. The notation for the probability vector described in the paragraph above also applies to the diagonal vector: D_p is the p-th block of the diagonal vector d. The diagonal vector is stored on disk to reduce the memory requirement of the solution process at the cost of increased disk I/O. The algorithm in Figure 4.2 uses a shared memory buffer $Dbox$ of size $n_{\max}$ short ints to read a block of diagonal vector from disk (line 14, *DiskIO Process*). An implementation of the symbolic out-of-core method based on the in-core diagonal storage would provide faster execution times at the cost of increased memory requirements.

The high-level structure of the algorithm, given in Figure 4.2, is that of a producer-consumer. The DiskIO Process acts as a producer while the Compute Process acts as a consumer. The two processes communicate using the shared memory buffers $Xbox_t$ and $Dbox$, and synchronise using a set of semaphores S_1 and S_2. In each execution of its inner `for` loop (lines $6-18$), the *DiskIO Process* reads the required vector block, X_q, and issues a Signal($\cdot$) operation. On receiving this signal, the *Compute Process* issues a return signal and then advances to carry out a unit of computation: the sub-

MVP $\check{A}_{pq}X_q$ (line 8, *Compute Process*). This activity (lines $5-9$, *Compute Process*) is repeated until all of the blocks in a block row have been read and their products have been accumulated in $\tilde{X}_p$. Once the whole p-th vector block has been updated, and the test for convergence has been performed, the *Compute Process* advances to the next phase and signals the *DiskIO Process*, which receives the signal and writes the new approximation for the p-th block to disk.

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

4.3.1 Notes on Improvements

It can be seen in the algorithm in Figure 4.2 that, in the i-th phase of an iteration, the two processes wait over different ($Xbox$) shared memory buffers, i.e. the *DiskIO Process* (line 11) waits over the shared buffer $Xbox_{\bar{t}}$, while the *Compute Process* (line 6) waits over the buffer $Xbox_t$. The two processes, however, wait over the same diagonal shared memory buffer, $Dbox$. Using one shared memory buffer to read the diagonal blocks can affect the concurrent execution of the two processes due to the requirement for the processes to synchronise on the single buffer. This can worsen the performance, but can be remedied in two ways.

Firstly, two shared memory buffers can be employed for the algorithm in Figure 4.2, in order for the two processes to concurrently read and consume the diagonal vector. At the cost of increased memory, this can improve the time performance of the algorithm. A point to note is that an additional shared buffer $Dbox$ will not necessarily cause an increase in the memory requirement for the symbolic out-of-core solution process. Increasing the number of blocks typically decreases n_{max}, the size of the largest matrix and vector block and, hence, decreases the overall memory requirement of the solution process. An additional $Dbox$ buffer can improve the overall time performance of the solution process due to possibly a reduced synchronisation cost. Consequently, the matrix can be partitioned into a higher number of blocks P to reduce the overall memory requirement of the out-of-core solu-

tion. It is difficult to make any claims, however, without an implementation and analysis of this approach.

Note, in Figure 4.2, that before moving to the next iteration, the *Compute Process* first consumes the shared buffer (see line 11; D_p^{-1} is stored in $Dbox$) and then performs a test for convergence. This imposes an additional delay in the synchronisation for the two processes. This situation can be improved by using the existing semaphores for the shared buffers $Xbox_t$ alone, and introducing another set of semaphores in the algorithm to synchronise on the diagonal buffer $Dbox$. As before, this addition of a semaphore must be experimented on before making any definitive claims.

4.4 First Implementation

The computation of the matrix-vector product (MVP) is the core operation of the iterative symbolic out-of-core algorithm. Its efficiency determines the overall performance of the algorithm. To implement the symbolic out-of-core algorithm given in Figure 4.2, the matrix is stored as an offset-labelled MTBDD data structure. To implement each sub-MVP, as required by line 8 of the *Compute Process* in Figure 4.2, we need to extract the matrix entries for a required matrix block from the MTBDD. Fast access to a required matrix block, therefore, is critical.

The offset-labelled MTBDD data structure has been described in Section 2.7, and we have seen that, because of its recursive nature, the MTBDD provides a natural decomposition of a matrix into its submatrices. Since an MTBDD is based on binary decisions, descending each level of the data structure splits the matrix into 4 submatrices. Hence, descending l levels, gives a decomposition into $(2^l)^2$ blocks. However, in order to produce an efficient representation, the MTBDD actually encodes a matrix over its potential state space, which typically includes many unreachable states. Furthermore, the distribution of these reachable states across the state space is unpredictable. Hence, descending l levels of the MTBDD actually results in blocks of varying and uneven sizes.

In order to obtain good performance from the symbolic out-of-core algorithm, the vector and matrix should be divided into partitions of equal sizes. Since the natural matrix decomposition of the MTBDDs results in blocks

of varying and uneven sizes, the associated variation in the sizes of vector blocks can seriously disrupt the overlap of numerical computation with disk I/O, and can consequently hinder its performance. Therefore, we decided to use equally sized partitions. To actualise this, we allocated an array of pointers, of the size of the number of partitions P. The i-th entry in this array points to a structure which contains references to all those subgraphs of the MTBDD which are required to perform a sub-MVP operation associated with the i-th matrix block.

The memory required to store this information is relatively small compared to the overall memory requirement of the solution process. This is because the array size is proportional to the number of blocks and is independent of the number of states. However, there is a substantial CPU cost associated with the initialisation of the array, and for the referencing of the block information during the MVP operation.

4.5 Second Implementation

We know from the previous section that the implementation of the MVP operation determines the overall performance of the symbolic out-of-core solution. In the first implementation of the algorithm, described above, we opted to use matrix (and vector) partitions of equal sizes. This was required to overlap the numerical computation with disk I/O. The implementation, however, did not provide the expected performance. To provide blocks of equal sizes, it was necessary to associate each matrix block with several different subgraphs of the MTBDD. Initialising and referencing this information required significant amount of additional time. The additional CPU requirement for extracting the required matrix entries overshadowed the benefit gained by reducing the disk I/O.

We experimented with another implementation of the symbolic out-of-core algorithm given in Figure 4.2. In this second implementation, we use the natural matrix partitioning provided by the MTBDD data structure. This is described as follows.

The MTBDD data structure provides a natural decomposition of a matrix into its submatrices. Descending l level of the data structure splits the matrix into $(2^l)^2$ blocks. For this second implementation, we select a value of l,

take $P = 2^l$, and use the natural decomposition of the matrix given by the MTBDD. To access each block we simply need to store a pointer to the relevant node of the offset-labelled MTBDD. One possible scheme would be to use a $P \times P$ array of pointers. However, as l grows larger, many of the matrix blocks are empty. Hence, in our implementation, we store the information in a sparse format based on the compact MSR scheme. Using this approach, as required by the inner loop of the *Compute Process* in Figure 4.2, the matrix blocks can be accessed in an efficient manner.

4.6 Experimental Results

In this section, we present experimental results collected from the two implementations of the symbolic out-of-core algorithm. The algorithms have been implemented on an UltraSPARC-II 440MHz CPU machine running SunOS 5.8 with 512MB RAM, and a 6GB local disk. We tested the symbolic implementations on three benchmark models. These are: a flexible manufacturing system (FMS) [CT93], a Kanban system [CT96] and a cyclic server Polling system [IT90]. The models were generated using the tool PRISM [KNP04a] (see Appendix A).

This section is organised as follows. In Section 4.6.1, we compare the two implementations of the symbolic out-of-core algorithm using the time per iteration results. In Section 4.6.2, we analyse the memory and time properties of the second implementation. In Section 4.6.3, we present a heuristic for a priori selection of the number of blocks P that the iteration vector is partitioned into. Finally, in Section 4.6.4, we compare the symbolic out-of-core method with the explicit out-of-core methods.

4.6.1 The Two Symbolic Implementations

We first compare the time per iteration results for the two implementations of the symbolic out-of-core algorithm. Table 4.1 summarises these results for the Kanban, FMS and polling system case studies. The parameter k in column 2 denotes the number of tokens in the Kanban and FMS models, and the number of stations in the polling system models. Column 3 lists the resulting number of reachable states in the CTMC matrices, and Column 4

Model	k	States	a/n	Blocks (P)		Time (sec/it)		Iter.		MB
		(n)		*1st*	*2nd*	*1st*	*2nd*	*1st*	*2nd*	per π
FMS	8	4,459,455	8.6	5	2^{23}	25.41	10.4	1255	1,245	34
	9	11,058,190	8.9	5	2^{23}	107.8	35.9	1424	1,416	84
	10	25,397,658	9.2	18	2^{23}	380	142	1594	1,591	194
	11	54,682,992	9.5	36	2^{23}	1,132	708	1763	1,770	417
	12	111,414,940	9.7	–	2^{23}	–	1,554	–	2174	850
	13	216,427,680	9.9	–	2^{23}	–	3,428	–	> 50	1,651
Kanban	5	2,546,432	9.6	4	2^{10}	11.5	4.5	565	532	20
system	6	11,261,376	10.3	4	2^{10}	49.4	22.6	767	717	86
	7	41,644,800	10.8	8	2^{10}	215	143	1003	924	317
	8	133,865,325	11.3	25	2^{13}	1,074	601	1091	1,151	1,004
Polling	18	7,077,888	9.8	4	2^5	21	10.5	311	302	54
system	19	14,942,208	10.3	4	2^5	53	23.8	317	315	114
	20	31,457,280	10.8	8	2^5	110	52	325	328	240
	21	66,060,288	11.3	16	2^5	364	177	332	340	504
	22	138,412,032	11.8	32	2^5	795	374	341	353	1,056

Table 4.1: Out-of-core numerical solution times for steady-state solution

gives the average number of off-diagonal entries per row.

The number of blocks, P, each vector is partitioned into are recorded in columns $5-6$. The number of blocks for the first implementation are listed under "1st" and for the second implementation are listed under "2nd". Accordingly, a matrix is divided into P^2 blocks. As explained in Section 4.4, the first implementation of the algorithm selects matrix (or vector) partitions of equal sizes based on the actual state space (number of reachable states). The number of partitions for the second implementation are based on potential state space, which typically includes many unreachable states. This explains the significant difference between the number of blocks for the two implementations. For both implementations, these value for P were selected with two considerations. Firstly, for smaller models, we collected timing results for a range of values of P and used this information to select the number of blocks for larger models. Secondly, we ensured that the amount of required RAM (corresponding to a selected value of the number P) for the solution of a particular CTMC should remain within the RAM available in the workstation. We will discuss this issue further in Section 4.6.3.

Note that, for the number of blocks reported in the table for the second implementation, we used 2^l notation in column 5, where l is the number of levels in an MTBDD. Note also that the number of blocks for the polling systems are relatively smaller compared to the Kanban and FMS systems. This is due to the fact that the Polling system is very structured compared to the Kanban and FMS systems and hence it results in a smaller MTBDD representation. We observe that, in general, the memory requirement of the solution process can be decreased by increasing the number of blocks.

The run times per iteration of the two symbolic out-of-core implementations are recorded in column $7-8$. The symbol '–' in column 7 against a row indicates that the system has not been solved using the first implementation on this workstation. All listed run times are in wall clock time. A first glance of the two columns confirms that the second implementation is significantly faster than the first.

Column $9-10$ give the number of iterations the iterative computations for the two implementations took to converge, using the criterion given by Equation (2.16) on Page 18. A steady pattern of convergence for all three models can be observed. The last column indicates the amount of memory required to store the probability vector (n doubles). Since the vector is stored on disk, this value also corresponds to the file size for the iteration vector.

The first case study in Table 4.1 is the flexible manufacturing system (FMS). The largest model solved in this case using the first implementation is for $k = 11$ with 54 million states; the solution took 24 days to complete. The same model, FMS ($k = 11$), solved using the second implementation took over 14 days to complete. The largest FMS model solved using the second implementation is for $k = 12$ with over 111 million states. The solution took 39 days to complete. The largest FMS model scheduled using the out-of-core solution method is for $k = 13$ with over 216 million states. The run times for this model are taken for 50 iterations; we were unable to wait for its convergence, and hence the total number of iterations is not reported in the table. The second implementation resulted in excessively large runtime per iterations for $k = 12, 13$, and therefore, we have not collected the results for these larger FMS models.

The second case study is the Kanban system. The largest CTMC solved in this case has 133 million states and over 1.5 billion transitions. The first

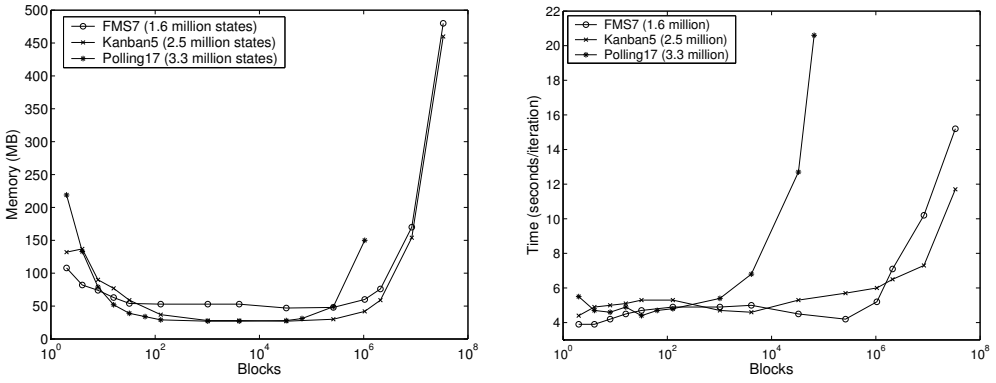
implementation in this case took 14 days to converge, while the second implementation took 8 days to converge. The final case study is the polling system, where the largest model solved is for $k = 22$ which contains 138 million states. The first implementation of the out-of-core solution for this system took just over 3 days to converge. The second implementation took 1.5 days to converge. Total solution times for the other models listed in Table 4.1 can be calculated using the time per iteration and the number of iterations.

We note in Table 4.1 that the second implementation is significantly faster than the first implementation of the symbolic out-of-core method. It is also evident from Table 4.1 that, for both the implementations, the FMS system has the highest solution times per iteration, given an equal number of states. The lowest solution times among the three reported models are attributed to the polling system. The FMS system is the least structured of the three models which equates to a large MTBDD to store it; the larger the MTBDD, the more time is required to perform its traversal. The polling system, on the other hand, is very structured and therefore results in a smaller MTBDD.

The memory requirements for the two implementations can be considered similar because it depends on the number of blocks a CTMC and the iteration vector are partitioned into. The second implementation of the symbolic out-of-core method, however, typically provides a speedup of approximately two over the first implementation. For the symbolic method hence, in future, we only consider the second implementation. A further analysis of the time and memory properties for the first implementation of the out-of-core method can be found in [KMNP02].

4.6.2 Further analysis of the Symbolic Solution

In Section 3.6.3 and Section 3.6.4, we have investigated the performance of the explicit matrix and complete out-of-core solutions by analysing the memory and time properties plotted against the number of blocks. In this section, we perform a similar analysis of the symbolic out-of-core method. We plot, in Figure 4.3, the memory and time properties against the number of blocks for the same three CTMCs, one from each model, as were used for the explicit out-of-core methods. To further explore the behaviour of the symbolic out-of-core method, in Figure 4.4, we plot similar properties for



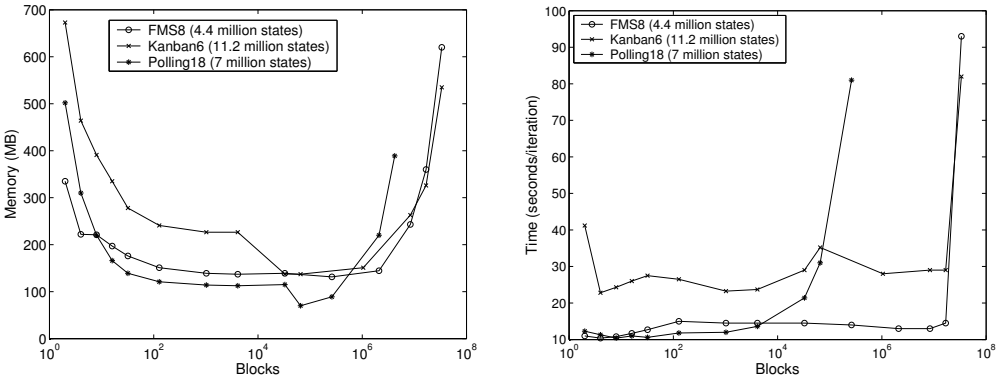
(a) Memory requirements vs. blocks (b) Time per iteration vs. blocks

Figure 4.3: The symbolic out-of-core method: space and time versus P

three larger CTMCs. In the following, we explain the plots for Figure 4.4; in the light of this discussion, Figure 4.3 should 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 4.4(a). We explain the figure with help of the plot for the Kanban system ($k = 6$). The memory required by the out-of-core solutions can typically be decreased by increasing the number of blocks. This is due to the fact that increasing the number of vector blocks usually causes a reduction in the size of the largest block to be kept in RAM, i.e. reduces $n_{\max}$. This is evident in the plot. The memory required for the case $P = 2$ is above 650MB. The increase in the number of blocks reduces the memory requirements for the Kanban system to nearly 140MB. Similar properties are evident for the other plots in Figure 4.4(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 4.3(b) and Figure 4.4(b), plotted against the number of vector blocks. We consider the plot for the Kanban system in Figure 4.4(b). Initially, for $P = 2$, the memory required (see Figure 4.4(a)) for the iteration vector is more than the available RAM. This causes thrashing



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

Figure 4.4: The symbolic out-of-core method: varying the number of blocks

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 4.5, 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 4.4(a).

Note that, for the symbolic implementations, we use the pseudo Gauss-Seidel method where we apply one Jacobi (inner) iteration on each subsystem, in a global Gauss-Seidel iterative structure (see Section 4.2). Increasing the number of blocks, therefore, typically causes a reduction in the required number of iterations for a CTMC to converge (number of iteration influences the total solution times).

Finally, we observe that, in Figure 4.3 and Figure 4.4, the number of blocks for the CTMCs are higher than the number of the reported (reach-

	FMS ($\times 10^{16}$)		Kanban($\times 10^{12}$)		Polling($\times 10^{6}$)	
	$k = 7$	$k = 8$	$k = 5$	$k = 6$	$k = 17$	$k = 18$
Unreachable states	2.02	11.85	2.82	33.24	4.46	9.44

Table 4.2: The number of unreachable states for the CTMCs

able) states. As explained in Section 4.5, the number of blocks, $P = 2^l$, for the symbolic out-of-core solution are selected by descending l levels of the MTBDD. Since an MTBDD actually encodes a CTMC matrix over its potential state space which includes many unreachable states, the number of blocks can exceed the number of reachable states. In Table 4.2, we list the number of unreachable states for the six CTMCs for which the time and memory behaviour is plotted in Figures 4.3 and 4.4. It is evident from the table that the number of unreachable states for the CTMCs are much larger than the number of respective reachable states. For example, the Kanban system ($k = 5$) has more than 2.5 million reachable states (see Figure 4.3), and more than 2.8 trillion unreachable states. These cases where the number of blocks exceed the number of (reachable) states are plotted to illustrate that (at this point) the memory and the time per iteration requirements for the solution method increase very rapidly, and to a very high value, and therefore, these cases are neither interesting nor practical.

4.6.3 A Priori Selection of P

We conclude here with our observation that the performance of the symbolic out-of-core solution method is dependent on the number of blocks, P, the matrix and vector are partitioned into. It follows from Figure 4.3 and Figure 4.4 that the symbolic out-of-core method delivers a satisfactory performance (i.e., low memory and time per iteration) for a wide range of values of this number P. A priori estimation of a value for number P which yields a desirable performance, along the lines of the ideas explained for the explicit methods in Section 3.6.5 (Page 62), is possible. A similar heuristic which estimates a value for number P based on the calculation of associated RAM for a number of values of P can be added to the phase responsible for setting up the offset-labelled MTBDD data structure.

Furthermore, it follows from the comparison of the two Figures 4.3 and 4.4, as we mentioned earlier that under different values of k (e.g. Kanban system, $k = 5, 6$) for each case study, the memory and time plots against the number of blocks display similar patterns. This can, in fact, be useful for predicting good choices of P for larger values of k.

4.6.4 In-Core and Out-of-Core Solutions

We now compare the performance of all of the in-core and out-of-core solution methods which we have discussed so far in this thesis. These include the solution methods presented in Chapter 3 which are based on the explicit storage of the CTMC matrix, and the symbolic solution methods presented in this chapter. These methods were tested for convergence using Equation (2.16) for $\epsilon = 10^{-6}$. We report the timing results in wall clock times.

Table 4.3 summarises results for the Kanban, FMS and Polling system case studies. As in the earlier sections, the first four columns list the model statistics. The time per iteration results for "Explicit" implementations are reported in columns 5–7 of Table 4.3: these include the standard in-core, where the matrix and the vector are kept in RAM; the matrix out-of-core (Section 3.4), where only the matrix is stored on disk and the vector is kept in RAM; and the complete out-of-core (Section 3.5), where both the matrix and the vector are stored on disk. The iterative method used for the explicit implementations is the block Gauss-Seidel method, and the resulting number of iterations is reported in column 8. The matrices for the explicit methods are stored in the compact MSR sparse storage scheme (Section 3.2). The scheme requires $4a + 3n$ bytes to store the whole matrix including the diagonal. The entries "a/n" in column 4 can be used to calculate the memory required to store the matrices.

We list the time per iteration results for "Symbolic" implementations in columns 9–10 of Table 4.3. These implementations include the in-core solution, where both the matrix and the iteration vector are stored in RAM, and the out-of-core (Section 4.5), where the matrix is kept in RAM and the vector is stored on disk. The run times for the in-core solution are collected using version 1.3.1 of the PRISM tool (see Appendix A). The out-of-core run times are for the second implementation of the symbolic algorithm. The matrix for these symbolic implementations is stored using the offset-labelled MTBDD

data structure, and the vector is kept as an array. The pseudo Gauss-Seidel iterative method (see Section 4.2) is used for the symbolic implementations, and the respective number of iterations is reported in column 11. The run times for FMS system ($k = 13$) are taken for 50 iterations; we were unable to wait for its convergence.

We first discuss the model sizes which can be solved on this particular workstation using a particular solution method. We note in Table 4.3 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 ($k = 18$) with approximately 7 million states. The in-core symbolic solution of column 9 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 ($k = 10$) with approximately 25 million states. To solve larger models, we have to turn to the out-of-core approaches.

The matrix out-of-core solution requires in-core storage for one iteration vector and two blocks of matrix. The memory required for these matrix blocks can, in general, be reduced by increasing the number of blocks. However, in this case, the limit on the largest solvable model exists due to the storage of the iteration vector. Pursuing the matrix out-of-core approach, the largest model solvable on the workstation is the Kanban system ($k = 7$) with approximately 41 million states. The out-of-core storage of both matrix and vector allows solution of even larger models. This is reported in column 7, and the largest model possible in this case is the Polling system ($k = 21$) with 66 million states. The limit in this case is the size of the available disk (6GB), although, using this approach, even larger models can be solved provided a larger disk is available.

The largest solvable model on the available machine is attributed to the symbolic out-of-core approach, i.e., the FMS system ($k = 13$) with 216 million states. The possibility exists because, in this case, the diagonal and iteration vectors are stored on disk and the CTMC matrix is compactly stored in-core, using the offset-labelled MTBDD data structure.

We now observe the relative speeds of the reported solution methods in Table 4.3. The matrix out-of-core and the symbolic in-core methods both provide a solution to the matrix storage problem, and hence, it is fair to

compare the run times for the two approaches. We note in Table 4.3, for the Polling and Kanban systems, that the run times per iteration for symbolic in-core (column 9) are faster than the matrix out-of-core method. However, for the FMS system, matrix out-of-core method provide better run times. Similarly, we compare the complete and the symbolic out-of-core approaches, because the two methods provide solution to both the matrix and the vector storage problems. 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.

It follows 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 (see Section 2.7.2). 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 this chapter with our observation that, for large models, the symbolic out-of-core solution provides the best overall results for the examples considered.

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	0.7	1.3	916
	7	1,639,440	8.3	1.1	1.7	3.8	966	3.4	3.2	1,079
	8	4,459,455	8.6	3.2	5.1	10.7	1,125	11.8	10.4	1,245
	9	11,058,190	8.9	–	24	51.8	1,287	37.7	35.9	1,416
	10	25,397,658	9.2	–	69	146	1,454	100	142	1,591
	11	54,682,992	9.5	–	–	374	1,624	–	708	1,770
	12	111,414,940	9.7	–	–	–	–	–	1,554	2174
	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	15.9	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	0.7	0.8	263
	16	1,572,864	8.8	1.1	1.9	2.9	33	1.6	2.0	276
	17	3,342,336	9.3	2.4	3.9	6.4	34	3.8	4.6	289
	18	7,077,888	9.8	5.5	15.8	20.4	34	8.1	10.5	302
	19	14,942,208	10.3	–	41	71	35	19.3	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 4.3: Comparing speeds for in-core and out-of-core methods

CHAPTER

FIVE

Improving Offset-Labelled MTBDDs

We have explored the out-of-core methods based on the sparse storage of CTMC matrices in Chapter 3, where we also introduced our complete (explicit) out-of-core method. We then turned our attention to the techniques based on the symbolic CTMC storage, and in this context, presented a new symbolic out-of-core solution method. We used the offset-labelled MTBDD data structure [KNP04b, Par02] to implement and investigate the symbolic out-of-core technique. In this chapter, we present modifications to the offset-labelled MTBDD data structure. Based on these modifications, we present and examine in-core and out-of-core implementations of the symbolic method for steady-state solution of CTMCs. The examination reveals that the modified approach renders substantial improvements over the original approach.

In Section 5.1, we present the motivations for the work presented in this chapter. In Section 5.2, we describe the underlying implementation of the offset-labelled MTBDDs. Using this description, in Section 5.3, we introduce and explain the modifications to the principal offset-labelled MTBDDs. Subsequently, we analyse the modified data structure using its in-core and out-of-core implementations. In Section 5.4, we study its behaviour when all data structures are stored in-core. In Section 5.5, we explore the performance of the symbolic out-of-core method based on this modified data structure, i.e. when the CTMC is stored in-core using the modified data structure and the iteration vector is kept on disk. To further improve performance of the symbolic out-of-core method, we propose some future work in Section 5.5.4.

5.1 Motivation

As discussed in Chapter 2, MTBDDs can often afford extremely compact storage for CTMC matrices and vectors and can be used to implement iterative numerical solution techniques such as Power and Jacobi. During the iterative solution phase, however, the MTBDD representation of the probability vector grows quickly and becomes impractical due to the increased number of distinct values in the vector. The solution usually adopted is to use offset-labelled MTBDDs [Par02, KNP04b] which combine MTBDD-based storage of the matrix with array-based storage of the solution vector.

The offset-labelled MTBDD approach retains the compact CTMC storage advantages of MTBDDs, and, for typical examples, can almost match the numerical solution speed of explicit approaches based on sparse CTMC storage. However, the offset-labelled MTBDD approach, as is the case for MTBDDs, does not allow the use of more efficient iterative methods such as the Gauss-Seidel method. Furthermore, in general (e.g. for less structured models), an iteration of the offset-labelled MTBDD-based iterative solution is still expensive compared to the explicit methods which allow direct (fast) access to the matrix entries.

In this chapter, we introduce some modifications to the offset-labelled MTBDD data structure which allow an efficient implementation of the Gauss-Seidel iterative method for the steady-state solution of CTMCs. These modifications also result in improved time and memory characteristics for the resulting offset-labelled MTBDD data structure.

5.2 Offset-Labelled MTBDDs

In Chapter 2, we have described the offset-labelled MTBDD data structure in some detail. In this section, we provide some additional information about the underlying implementation of the offset-labelled MTBDDs, which will be required in Section 5.3 where the modifications to the principal data structure are introduced. For explanation and completeness, we reiterate some information from Chapter 2 on offset-labelled MTBDDs in this section. Note that the (step-wise) optimisations to the offset-labelled MTBDDs described in this section (and its subsections) have been proposed by Parker [Par02] and

have been implemented in the PRISM tool version 1.3.1. The optimisations presented in Section 5.3 are the contribution of this thesis.

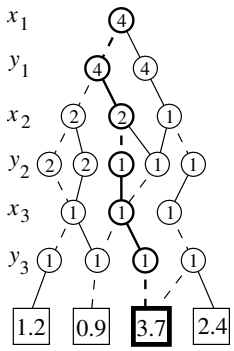
In order to explain the details of the implementation of offset-labelled MTBDDs, we introduce the following running example. We show an 8×8 matrix in Figure 5.1, its representation as an offset-labelled MTBDD, and a table which explains the encoding of the information. The matrix is assumed to be derived from the generator matrix of a CTMC. Note that, to preserve structure in the symbolic representation, the diagonal elements are stored separately as an array. Hence, the diagonal entries of the matrix in Figure 5.1 are all zero.

The offset-labelled MTBDD in Figure 5.1 represents a function over the six Boolean variables $x_1, y_1, x_2, y_2, x_3, y_3$. For a given valuation of these variables, the value of the function can be computed by tracing a path from the top of the MTBDD to the bottom, at each node taking the dotted edge if the associated Boolean variable is 0 and the solid edge if it is 1. The value can be read from the label of the terminal node reached. For example, if $(x_1, y_1, x_2, y_2, x_3, y_3) = (0, 1, 0, 1, 1, 0)$, the function returns 3.7 (as highlighted in the figure). However, If $(x_1, y_1, x_2, y_2, x_3, y_3) = (0, 0, 0, 0, 0, 0)$, the function returns zero because there is no such path. We indicate this in the figure, by listing a "-" in the corresponding "Path" entry for the first row in the table. Of course, the matrix is sparse. For explanation, we (only) list one zero entry in the table.

The matrix represented by the offset-labelled MTBDD in Figure 5.1 is encoded by the function (explained in the above paragraph) as follows. The x_i variables are for row indices, and the y_i variables are for column indices. Since these variables are all Boolean, but the row and column indices are integers in the range $0 \ldots 7$, the information has to be encoded. We assume the standard binary representation of integers for this purpose. Consider the matrix entry $(1, 6) = 3.7$. The row index is 1 so we code this as 001 ($x_1 = 0, x_2 = 0, x_3 = 1$). The column index is 6 so we code this as 110 ($y_1 = 1, y_2 = 1, y_3 = 0$). The x_i and y_i variables are interleaved in the MTBDD to reduce its size. The entry $(1, 6)$ is hence represented by the path 010110 which, as we have seen above, leads to the value 3.7.

The integer offsets on the nodes of the data structure are used to compute the actual row and column indices (in terms of reachable states only) of the

0	1.2	0	0	0	0	0.9	0
0.9	0	0	0	0	0	**3.7**	0
0	0	0	1.2	0.9	0	0	0
0	0	0.9	0	3.7	0	0	0
0	0	0	0	0	0	3.7	2.4
0	0	0	0	0	0	0	0
0	0	0	0	0.9	0	0	0
0	0	0	0	3.7	0	0	0



Matrix entry	Encoding						Path						Offsets						Reachable entry
	x_1	x_2	x_3	y_1	y_2	y_3	x_1	y_1	x_2	y_2	x_3	y_3	x_1	y_1	x_2	y_2	x_3	y_3	
$(0,0) = 0$	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	$(0,0) = 0$
$(0,1) = 1.2$	0	0	0	0	0	1	0	0	0	0	0	1	-	-	-	-	-	1	$(0,1) = 1.2$
$(0,6) = 0.9$	0	0	0	1	1	0	0	1	0	1	0	0	-	4	-	1	-	-	$(0,5) = 0.9$
$(1,0) = 0.9$	0	0	1	0	0	0	0	0	0	0	1	0	-	-	-	-	1	-	$(1,0) = 0.9$
$\mathbf{(1,6) = 3.7}$	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**-**	**4**	**-**	**1**	**1**	**-**	$\mathbf{(1,5) = 3.7}$
$(2,3) = 1.2$	0	1	0	0	1	1	0	0	1	1	0	1	-	-	2	2	-	1	$(2,3) = 1.2$
$(2,4) = 0.9$	0	1	0	1	0	0	0	1	1	0	0	0	-	4	2	-	-	-	$(2,4) = 0.9$
$(3,2) = 0.9$	0	1	1	0	1	0	0	0	1	1	1	0	-	-	2	2	1	-	$(3,2) = 0.9$
$(3,4) = 3.7$	0	1	1	1	0	0	0	1	1	0	1	0	-	4	2	-	1	-	$(3,4) = 3.7$
$(4,6) = 3.7$	1	0	0	1	1	0	1	1	0	1	0	0	4	4	-	1	-	-	$(4,5) = 3.7$
$(4,7) = 2.4$	1	0	0	1	1	1	1	1	0	1	0	1	4	4	-	1	-	1	$(4,6) = 2.4$
$(6,4) = 0.9$	1	1	0	1	0	0	1	1	1	0	0	0	4	4	1	-	-	-	$(5,4) = 0.9$
$(7,4) = 3.7$	1	1	1	1	0	0	1	1	1	0	1	0	4	4	1	-	1	-	$(6,4) = 3.7$

Figure 5.1: Representing an 8×8 matrix as an (offset-labelled) MTBDD

matrix entries because the potential state space can typically 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 5.1, state 5 (counting from state index zero) is not reachable. For the previous example of matrix entry $(1, 6)$, the actual row index is 1 and the actual column index is $4 + 1 = 5$. Therefore, the actual index for the matrix entry $(1, 6)$ is $(1, 5)$.

5.2.1 Speeding up the Graph Traversal

For the Power and Jacobi iterative methods, each iteration is based essentially on a matrix-vector product operation. This requires access to each matrix element, in any order, exactly once. When using the offset-labelled MTBDD data structure to store a CTMC matrix (see Figure 5.1), the matrix entries can be accessed via a single depth-first traversal of the data structure since each matrix element corresponds to a path through the MTBDD. The overhead associated with this makes the process slower than the equivalent operation with sparse matrices, since it is significantly faster to read the matrix entries directly from an array-based data structure. Significant optimisation of the process has been realised in [KNP04b, Par02] by replacing bottom portions of the offset-labelled MTBDD graph with their explicit array-based representations. This is explained next.

It can be seen in Figure 5.1 that each non-terminal node of the offset-labelled MTBDD (also true for an MTBDD) represents a submatrix of the matrix represented by the whole MTBDD. For example, x_2 nodes and x_3 nodes represent 4×4 and 2×2 submatrices, respectively, of the 8×8 matrix. To explain this, we reproduce the matrix and its offset-labelled MTBDD representation in Figure 5.2(a) and Figure 5.2(b), respectively. The CTMC matrix in this figure consists of four 4×4 submatrices, or sixteen 2×2 submatrices. We observe in the figure that the nodes near the bottom of the offset-labelled MTBDD, in particular, are visited many times during one traversal (i.e. one iteration of the solution method). It is much faster to extract entries of the CTMC matrix if some of these nodes are replaced with an explicit representation of their corresponding submatrix, since this will eliminate the need to traverse the nodes below this point. These modifications

to offset-labelled MTBDDs are implemented as follows.

As mentioned in Section 2.7.3, a *level* of an MTBDD is an adjacent pair of rows of nodes; counting levels from the top of the MTBDD, level i contains all the x_i and y_i nodes. The total number of levels is denoted l_{total}. The modifications proposed in the above paragraph are made by selecting a value $l_{sm} \leq l_{total}$ and replacing the x_i nodes in layer $l_{total} - l_{sm} + 1$ with an explicit, sparse representation of their corresponding submatrices. This means that nodes in the bottom l_{sm} levels do not need to be traversed and can be removed entirely. The storage scheme employed for this explicit storage is the *compressed sparse row* (CSR) sparse matrix scheme. Its implementation in this case uses three arrays, `Val` and `Col`, which contain the value and column index, respectively, of each non-zero matrix entry, stored row by row, and `Starts`, which contains indices into `Val` and `Col`, indicating where the entries for each row are stored. See Section 2.6 for a description of the CSR scheme.

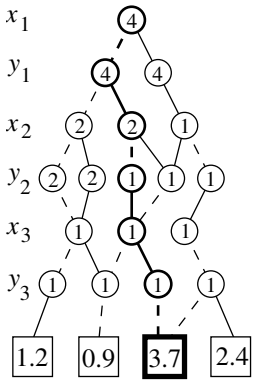
The optimisation described in the above paragraph is illustrated in Figure 5.2(c) using $l_{sm} = 1$ ($l_{total} = 3$). The three arrays for the CSR scheme are denoted `V`, `C` and `S`, respectively. The offset-labelled MTBDD path for the matrix entry 3.7 with the actual index $(1, 5)$ is highlighted in the figure. This optimisation of the offset-labelled MTBDD data structure has been implemented in the tool PRISM version 1.3.1. Generally, as l_{sm} is increased, the time for each iteration of numerical solution (i.e. a single traversal) decreases, but the required memory increases. The strategy used in the tool is to choose l_{sm} as high as possible without exceeding some predefined memory limit on the aggregate memory required to explicitly store all the submatrices. This memory limit can be set in PRISM by the user. The default limit is set at `1MB`.

5.2.2 A Three-Layered Perspective

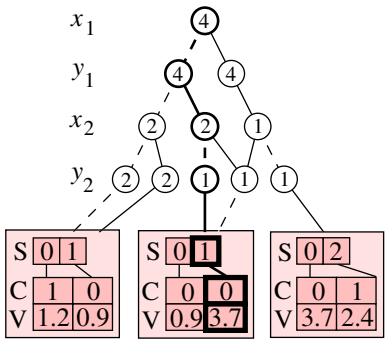
A problem with the MTBDD approach described above is that, although the Jacobi iterative method can be efficiently implemented, Gauss-Seidel cannot because it requires row-wise access to matrix entries. A depth-first traversal of the MTBDD does not allow matrix entries to be extracted in this order. Of course, it would be possible to access each element of each row individually, going from top to bottom of the MTBDD each time, but

0	1.2	0	0	0	0	0.9	0
0.9	0	0	0	0	0	**3.7**	0
0	0	0	1.2	0.9	0	0	0
0	0	0.9	0	3.7	0	0	0
0	0	0	0	0	0	3.7	2.4
0	0	0	0	0	0	0	0
0	0	0	0	0.9	0	0	0
0	0	0	0	3.7	0	0	0

(a)



(b)

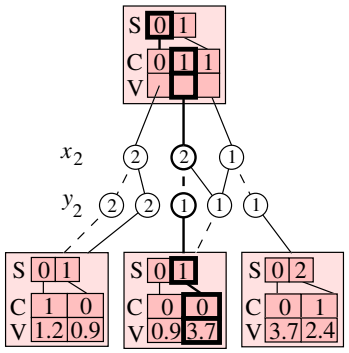


(c)

Figure 5.2: An Optimisation of MTBDDs

0	1.2	0	0	0	0	0.9	0
0.9	0	0	0	0	0	**3.7**	0
0	0	0	1.2	0.9	0	0	0
0	0	0.9	0	3.7	0	0	0
0	0	0	0	0	0	3.7	2.4
0	0	0	0	0	0	0	0
0	0	0	0	0.9	0	0	0
0	0	0	0	3.7	0	0	0

(a)



(b)

Figure 5.3: Optimisations to the MTBDD storage scheme

this would be very inefficient. In [Par02], a compromise between Jacobi and Gauss-Seidel called Pseudo Gauss-Seidel was implemented for the offset-labelled MTBDD data structure . We have described the pseudo Gauss-Seidel algorithm in Section 4.2 and have used this method for the implementation of the symbolic out-of-core solution method presented in Chapter 4. We describe now the modifications to offset-labelled MTBDDs, proposed and implemented by Parker [Par02], which enabled the implementation of pseudo Gauss-Seidel iterative method with the data structure.

We know from our earlier discussions that MTBDDs allow convenient access to matrix blocks. Descending one level from the top of an MTBDD splits the matrix which it represents into 4 blocks. Descending l_b levels, for

some $l_b \leq l_{total}$, gives a decomposition into P^2 blocks, where $P = 2^{l_b}$. If pointers to the nodes which represent these blocks are stored explicitly, these blocks can be accessed quickly without having to traverse the top part of the MTBDD. For large l_b, many of the matrix blocks will actually be empty so a sparse matrix scheme can be used. Like for the bottom layer, the CSR scheme is used. However, in this case, node pointers are stored in the array `Val` instead of numerical values. Nodes in the top l_b levels of the MTBDD (like the bottom l_{sm} levels) can now be removed entirely. We illustrate these modifications to the offset-labelled MTBDD data structure in Figure 5.3(b) for $l_b = 1$. This implies $P = 2^{l_b} = 2$. The CTMC matrix is reproduced in Figure 5.3(a). A solid line is used to emphasize the top layer decomposition of the matrix into four 4×4 submatrices. The dashed lines show the bottom-layer submatrices of the offset-labelled MTBDD. Note that, since one of the four ($P^2 = 4$) top-layer matrix blocks is empty, only 3 node pointers are stored. An additional array `Offsets` is required to store the (global) index of the first row of each block in terms of reachable states. This information replaces the offsets on the top layer of MTBDD nodes which have now become obsolete.

It is evident from Figure 5.3(b) that the divisions of a matrix into *blocks* by descending from the top of the MTBDD, and into *submatrices*, by ascending from the bottom of the MTBDD can happily coexist, provided that the top and bottom layers do not overlap, i.e. $l_{sm} + l_b \leq l_{total}$. Note that, although the two terms are in general interchangeable, for convenience, from hereon, we will consistently distinguish between "submatrices" and "blocks" in this way.

The modifications described above facilitate efficient access to the matrix elements, if not by individual rows, then at least by rows of blocks. This permits an efficient implementation of the pseudo Gauss-Seidel method. In an implementation of the pseudo Gauss-Seidel method based on this offset-labelled MTBDD data structure, the value of l_b can be increased to reduce both the number of iterations required to converge and the amount of memory for the vector storage, although not to the same extent as the standard Gauss-Seidel. In this case, increasing l_b does, however, also lead to exponential growth in the amount of memory required for storage of the top layer of the data structure. In version 1.3.1 of the PRISM tool, the parameter l_b

is selected as high as possible such that the memory required to store top layer of the data structure does not exceed some limit. As for the bottom layer storage, the default value for this is set at 1MB. See [Par02] for further implementation details and analysis of these optimisations.

5.3 Modifying Offset-Labelled MTBDDs

We now propose improvements to the offset-labelled MTBDD techniques described in the previous section. Recall that the offset-labelled MTBDD is a three layered data structure (see Figure 5.3). The top layer of the data structure consists of a blocked matrix which is stored using the compressed sparse row (CSR) format. Each entry of this blocked matrix is a pointer to an offset-labelled MTBDD. The middle layer of the data structure – an offset-labelled MTBDD – consists of nodes connected in a directed acyclic graph (DAG). The bottom layer of this data structure consists of submatrices which are reached by traversing the nodes in the middle layer DAG. Each of these submatrices is stored separately using the CSR format. A high-level, simplified perspective of offset-labelled MTBDDs can also be seen in Chapter 2, Figure 2.6 on Page 30.

The storage for the three layers of the offset-labelled MTBDD data structure has been explained in Section 5.2. However, implementation of the top layer storage which represents the blocked matrix needs further explanation. As before, assume that the top layer constitutes l_b levels of the MTBDD, that $P = 2^{l_b}$, and that the sparse storage scheme used is CSR. Assume also that only $nnzP_{mat}$ of the P^2 matrix blocks are non-empty. Hence, the top layer of the data structure consists of the following four arrays:

- The array `Val` of $nnzP_{mat}$ pointers. Each entry of this array points to an offset-labelled MTBDD node.
- The array `Col` of $nnzP_{mat}$ integers. The i-th entry in this array stores the column index of the i-th entry in the array `Val`.
- The array `Starts` of P integers. The i-th entry in this array contains the index in the arrays `Val` (and `Col`) of the beginning of the i-th row of blocks.

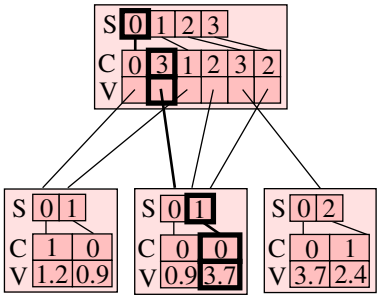
- The array `Offsets` of P integers. This array gives us the (global) index of the first row of each block in terms of reachable states.

The bottleneck in the storage of the top layer is the two arrays `Starts` and `Offsets`, the size of which ($4P$ bytes each) is exponential in l_b. Since the matrix is sparse, structured, and contains many unreachable states, many of the P rows of matrix blocks are empty. Therefore, it is more efficient to use these two arrays of length $nnzP$, where $nnzP$ is the number of non-zero rows of blocks in the matrix (or, equivalently, blocks in the vector). The relationship between $nnzP$ and $nnzP_{mat}$ is analogous to the relationship between n (number of CTMC states, or rows in the matrix) and a (number of nonzero entries in the off-diagonal matrix). Furthermore, we know from Section 2.6.2 that, in general, the maximum number of entries per row of a generator matrix is small, and is limited by the maximum number of transitions leaving a state. If this number does not exceed 128, the number of nonzero matrix entries per row can be represented as a char (a byte); see [DS98b, KH99, BH01, KM02]. We have exploited this property in the experiments presented in Chapter 3. Similarly, based on our empirical results, the maximum number of nonzero blocks per row of matrix blocks does not exceed 128, and hence, we use a char to represent the number of nonzero blocks in a row of matrix blocks. Essentially, we use the compact MSR format (see Section 3.2), instead of CSR to store the top layer of the data structure. The resulting top layer storage comprises the following:

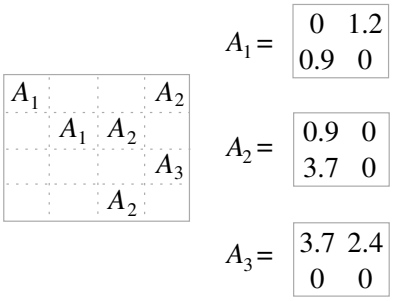
- An array `Val_dist` of d pointers. This array stores each distinct offset-labelled MTBDD node pointer once only. The number of distinct MTBDD node pointers, d, depends on the model and is typically small.

- The array `Col` of $nnzP_{mat}$ integers. Each element of this array stores the column index of a top-layer block of the matrix (an offset-labelled MTBDD node), and the index into the array `Val_dist` for this matrix block. This index into the array `Val_dist` is stored in the spare bits of `Col[i]`, eradicating the need for the array `Val` entirely (see Section 3.2 for details).

- The array `Starts` of $nnzP$ unsigned char. The i-th entry of this array now contains the number of nonzero blocks in the i-th row of matrix



(a) The two-layer storage



(b) Block partitioning of the 8×8 matrix

Figure 5.4: The modified offset-labelled MTBDD storage for the CTMC

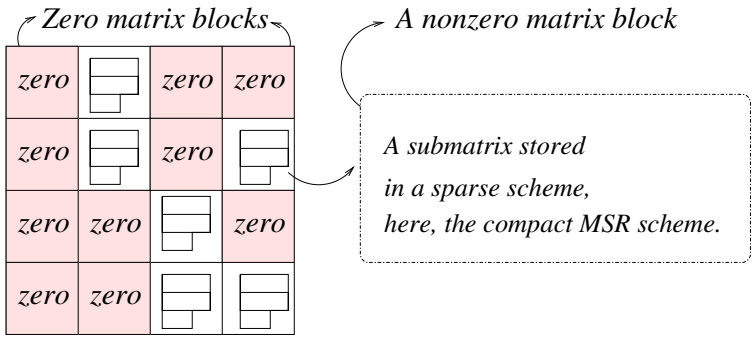


Figure 5.5: A modified offset-labelled MTBDD: another perspective

blocks. This is equivalent to the information stored previously, but requires only one byte per entry instead of four bytes.

- The array `Offsets` of $nnzP$ integers.

We note that the memory required for the modified data structure detailed above is no longer exponential in the number of levels l_b. To further reduce the memory requirements of the data structure, we use the compact MSR scheme instead of CSR to store the bottom layer of the data structure.

Provided that the total memory required for the storage of the matrix in the modified offset-labelled MTBDDs remains affordable, as we will see in Section 5.4, it is possible to select much larger values for l_b and l_{sm}, and hence to remove the middle-layer storage of the original offset-labelled MTBDD data structure, i.e., it is possible now with these modifications to remove the middle-layer directed acyclic graph. We choose values such that $l_b + l_{sm} = l_{total}$, i.e. the *blocks* indexed by the top layer and the *submatrices* described on the bottom layer coincide, the middle layer of the data structure being removed entirely. Consequently, the resulting data structure consists of two layers of sparse storage. This is illustrated in Figure 5.4(a), where we show the modified offset-labelled MTBDD data structure for the running example of Figures 5.2 and 5.3. For clarity of presentation, the sparse matrix scheme used for both the top and bottom layers in this figure is actually CSR, rather than compact MSR. Figure 5.4(b) illustrates the resulting partitioning of the 8×8 example matrix, emphasizing the distinct blocks in the CTMC matrix.

Another view of the modified offset-labelled MTBDD data structure is

depicted in Figure 5.5, where a matrix consists of a number of blocks. The top layer of the data structure stores the blocked sparse matrix using the compact MSR format. Each of the bottom layer submatrices is also stored using compact MSR format. Note that, although the new data structure is made up entirely of sparse matrix storage, it should still be seen as a symbolic approach. The data structure is constructed directly from the MTBDD representation and is completely reliant on the exploitation of regularity that this provides.

5.3.1 Implementing Gauss-Seidel

Using the modified offset-labelled MTBDD data structure described above, we are now able to obtain an efficient implementation of the Gauss-Seidel iterative method. This is so because the modified data structure provides efficient access both to each row of matrix blocks, and to each individual row within these blocks. Using this data structure, we have employed the block Gauss-Seidel algorithm given in Chapter 3 to implement in-core and out-of-core solutions. Results for these implementations are presented in Section 5.4 and Section 5.5.

5.4 Experimental Results: In-Core

In this section, we analyse the performance of the modified offset-labelled MTBDD data structure for the in-core steady-state numerical solution of CTMCs. The term "in-core" emphasises that all data structures, the matrix and the vector, are kept in RAM. We performed steady-state probability computation on the three sets of benchmark CTMCs: the flexible manufacturing system (FMS) [CT93], the Kanban system [CT96] and the cyclic server polling system [IT90]. These models were generated using the probabilistic model checker PRISM; see Appendix A and B for details. Experiments were run on a 440MHz 512MB UltraSPARC-II workstation running SunOS 5.8. Iterative methods were tested for the convergence criterion given by Equation (2.16) for $\varepsilon = 10^{-6}$.

In this section, we first analyse the memory and time properties of the modified offset-labelled MTBDD data structure against a range of values for

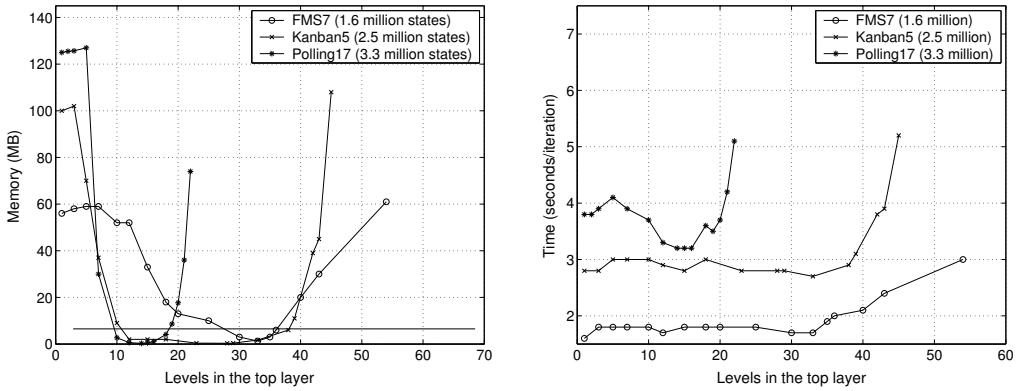
the number of block levels l_b. From this analysis, we derive a heuristic for a priori selection of this parameter. Using this heuristic, we present the numerical solution times for the modified data structure in Section 5.4.2. These timing results are compared with the original offset-labelled MTBDDs and an explicit sparse implementation. Finally, in Section 5.4.3, we investigate the relative memory properties of the two symbolic data structures. In this chapter, we will use the term "two-layer" and "modified offset-labelled MTBDD data structure" interchangeably. In all cases, the time taken to generate the two symbolic representations of a CTMC were negligible compared to the numerical solution times, and hence we concentrate in this chapter on the latter.

5.4.1 A Good Choice for the Parameter l_b

The selection of the parameter l_b determines l_{sm} and controls the size of the top and bottom layers of the modified offset-labelled MTBDDs. The parameter is hence a crucial factor for its performance. In this section, we analyse the issue in more detail. Although determining the optimal value for l_b is likely to be expensive or infeasible, we seek a heuristic which provides good performance.

An MTBDD comprises a total number of levels l_{total}. As has been explained in Section 5.2, for the original offset-labelled MTBDDs, the CTMC matrix is decomposed into 2^{l_b} blocks by selecting a value for parameter $l_b \leq l_{total}$. The value for parameter l_{sm} is selected such that $l_b + l_{sm} \leq l_{total}$, and all the MTBDD nodes at the level $l_{total} - l_{sm} + 1$ are replaced with an explicit sparse representations of their corresponding submatrices. In the implementation of the offset-labelled MTBDDs for version 1.3.1 of the PRISM tool, these two parameters, l_b and l_{sm}, are selected such that the storage for the top and the bottom layers do not exceed some fixed limits. The default values are set at 1MB for both parameters.

For the modified offset-labelled MTBDD data structure, we collected memory and time per iteration results for the same three sets of CTMCs against a range of values for the parameter l_b. In this case, $l_b + l_{sm} = l_{total}$. Some of these results are plotted in Figure 5.6. We first explain Figure 5.6(a), where we have plotted the memory for the off-diagonal matrix storage against the parameter l_b. The number l_b also governs the parameter n_{max}, which is



(a) Off-diagonal Memory (b) Time per iteration

Figure 5.6: Performance: behaviour against l_b

the size of the matrix block consisting of the largest number of rows. Hence, in the plotted memory, we have also included the RAM to store the additional iteration vector of size n_{max} doubles, which is required to hold the sub-MVPs during an iteration.

Consider the plot for the Polling($k = 17$) system. The total number of MTBDD levels, l_{total}, for this model is 23 (see Table 5.1). Initially, for $l_b = 1$, the required memory is above 120MB. In this case, $l_{sm} = l_{total} - 1 = 22$, and therefore the resulting CTMC storage consists of the sparse storage of four large submatrices. Model structure is not exploited in this case. Increasing the parameter l_b typically increases the memory required for the top-layer block sparse storage. However, in this case, there is also a decrease in the memory required to store the submatrices at the bottom layer. Increasing l_b to 16 decreases the memory to below 1MB. From this point on the memory in the plot increases with an increase in the number of levels, in particular as the parameter l_b approaches the total number of MTBDD levels, l_{total}. In these cases the top-layer block storage dominates the overall memory. Similar behaviours are observed for the Kanban and FMS plots except that the changes in these plots are relatively slow due to a larger l_{total} (see Table 5.1).

The time per iteration properties of the modified offset-labelled MTBDDs are plotted in Figure 5.6(b) against the parameter l_b. Consider again the plot for the Polling system. The time per iteration for the polling CTMC

fluctuates until it reaches the minimum of 3.2 seconds. The time from this point on increases as the number of block levels l_b approaches l_{total}. We note similar behaviours in the Kanban and FMS plots except that the fluctuations in these plots last longer due to a larger l_{total}.

We now summarise our findings from these experiments. We observe that, for all three examples, the minimum and maximum values of l_b result in very high memory usage. This is unsurprising since, in these extreme cases, the sparse matrix storage for either the bottom or top layers, respectively, constitutes almost the whole of the data structure. In these cases, none of the regularity and compactness of MTBDDs is exploited, and hence the memory usage is high. However, we see that, for values of l_b in the middle of this range, we can easily find a compromise between storage for the top and bottom layers which gives a dramatic drop in memory. For times per iteration (Figure 5.6(b)), we see that there are some fluctuations as l_b is varied and a consistent increase as it reaches its maximum. Overall, though, the variations in time are not as significant as for the memory. Based on these statistics, we adopt the heuristic $l_b = 0.6 \times l_{total}$ for the in-core implementation of the modified offset-labelled MTBDD data structure.

Finally, note that, since the original offset-labelled MTBDDs employ the pseudo Gauss-Seidel method, the parameter l_b also governs the number of iterations for the solution process to converge. Typically, an increase in the value of l_b causes a decrease in the number of iterations. However, the number of iterations for the modified offset-labelled MTBDDs are independent of the parameter l_b, for it can employ the standard Gauss-Seidel iterative method.

5.4.2 Speed of Numerical Solution

We now compare the solution based on the modified offset-labelled MTBDD data structure against the implementations of the original offset-labelled

	FMS($k = 7$)	Kanban($k = 5$)	Polling($k = 17$)
l_{total}	55	48	23

Table 5.1: The total number of levels, l_{total}, for the three CTMCs

Model	k	States (n)	a/n	Time/iter. (secs)			Iterations		Total time (secs)		
				Sparse (GS)	MTBDD (PGS)	Two-layer (GS)	PGS	GS	Sparse (GS)	MTBDD (PGS)	Two-layer (GS)
FMS	6	537,768	7.8	0.3	0.72	0.50	1,027	812	244	739	406
	7	1,639,440	8.3	1.1	3.44	1.61	1,207	966	1,063	4,152	1,555
	8	4,459,455	8.6	3.2	11.8	4.68	1,392	1,125	3,600	16,426	5,265
	9	11,058,190	8.9	–	37.7	12.3	1,581	1,287	–	59,604	15,830
	10	25,397,658	9.2	–	100	29.2	1,775	1,454	–	177,500	42,457
Kanban	4	454,475	8.8	0.3	0.52	0.44	414	323	96.9	215	142
	5	2,546,432	9.6	1.8	3.13	2.71	590	461	830	1,847	1,249
	6	11,261,376	10.3	–	15.9	12.9	794	622	–	12,625	8,024
Polling system	15	737,280	8.3	0.5	0.69	0.62	179	32	16.0	124	19.8
	16	1,572,864	8.8	1.1	1.57	1.45	196	33	36.3	308	47.9
	17	3,342,336	9.3	2.4	3.78	3.23	213	34	81.6	805	110
	18	7,077,888	9.8	5.5	8.11	7.17	229	34	187	1,857	244
	19	14,942,208	10.3	–	19.3	16.3	255	35	–	4,922	571
	20	31,457,280	10.8	–	–	36.8	–	36	–	–	1,325

Table 5.2: Timing results: A comparison with existing implementations

MTBDDs (as implemented in PRISM version 1.3.1) and an explicit method. For the explicit implementation, we use the compact MSR scheme to store the CTMC. Table 5.2 reports timing results for these implementations. Results for the compact MSR based explicit method are reported under "Sparse". Results for the original and the modified offset-labelled MTBDD data structure are listed under "MTBDD" and "Two-layer" respectively. In each case, we use the most efficient numerical solution method available, i.e. Gauss-Seidel for the compact MSR and the modified offset-labelled MTBDDs, and Pseudo Gauss-Seidel for the original offset-labelled MTBDDS.

The first four columns in Table 5.2 give details of each CTMC used, its size n (reachable states) and average number of non-zeros per row (a/n). Column 2 gives the values for the model parameter k, see Appendix B for details. For each implementation, we give the average time per iteration, the number of iterations and the total time. Columns $5-7$ and columns $10-12$ list time per iteration and total times for the three solution methods. The number of iterations for the original offset-labelled MTBDDs (PGS) are enumerated in column 8, and for the other two solutions (GS) are listed in column 9. For pseudo Gauss-Seidel, the number of iterations is dependent on the number of blocks the CTMC matrix is decomposed into. The number of Gauss-Seidel iterations for a particular model remains unchanged.

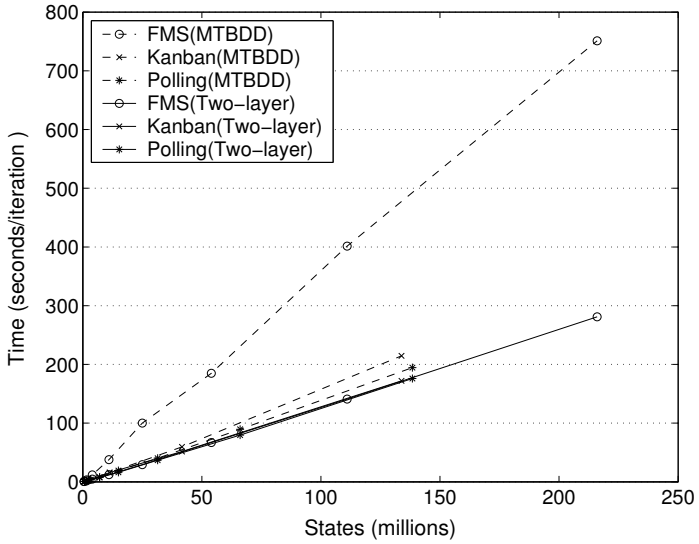


Figure 5.7: Times per iteration results plotted against number of states

To give a better indication of the trends in these statistics of Table 5.2, for the two symbolic methods we plot time per iteration and total time against the number of states in Figures 5.7 and Figure 5.8, respectively. Dashed lines are used to draw the results for the original offset-labelled MTBDDs, and are tagged "MTBDD". The results for modified offset-labelled MTBDDs are drawn in solid lines and are tagged with "Two-layer". For these plots we have collected results on a machine with an equivalent CPU (440MHz) but with more RAM (5GB), in order to illustrate trends over a larger range of state spaces.

Our first observation in Table 5.2 is that the average time per iteration for the modified offset-labelled MTBDDs presents an improvement over the original MTBDD implementation. The principal reason for this is that the former approach is based on the array-based storage, and therefore, it avoids the traversal of MTBDD nodes in the middle layer of the data structure. This also means that the modified approach behaves more like a sparse storage based method and provides a much more consistent performance across the three examples. This is obvious in Figure 5.7, where all three plots show an equal time per iteration against the number of states, i.e. the three time plots are on the same line. Original offset-labelled MTBDDs, on the other hand,

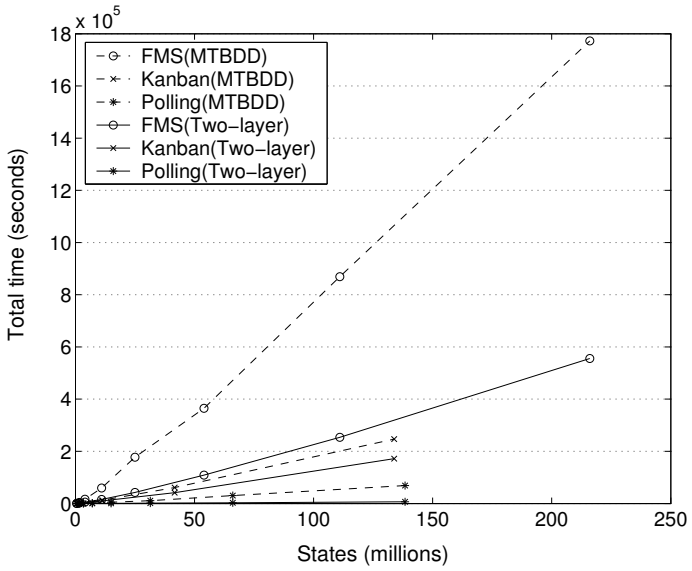


Figure 5.8: Total times plotted against number of states

are more reliant on structure in the CTMCs and, therefore, their performance varies across the three models; see dashed lines in Figure 5.7 where the time results against the number of states for all three models have different slopes. The FMS model is the least structured among the three case studies which equates for its higher times; Polling is the fastest because it is the most structured model among the three. Consequently, the FMS model shows greater improvements between the two symbolic approaches; the run times for the modified data structure (for large models) are at least three times faster than the original offset-labelled MTBDDs. We also note, in Figure 5.8 and in Table 5.2, that the total solution times for the modified offset-labelled MTBDDs show an even more impressive improvement. This is due to the fact that Gauss-Seidel typically requires a smaller number of iterations to converge compared to the pseudo Gauss-Seidel method.

Making a comparison with sparse matrices, we see in Table 5.2 that the solution times for the modified offset-labelled MTBDDs are now much closer than the times for MTBDDs were previously. We also note that the modified approach can handle CTMCs approximately an order of magnitude larger than sparse matrices, due to the relatively compact memory requirements. Furthermore, it can also be applied to slightly larger CTMCs than MTBDDs

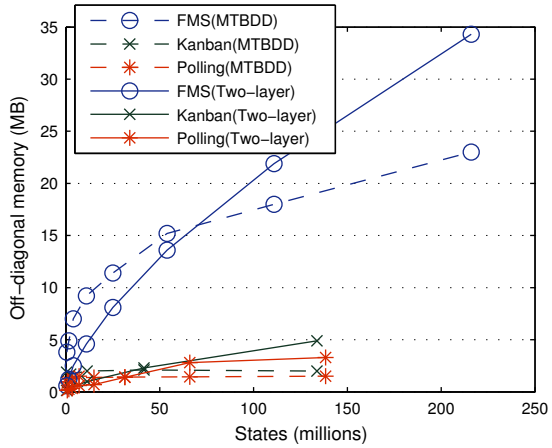
can. This is because its implementation uses two bytes for each element of the diagonal vector, instead of eight bytes, exploiting the small number of distinct values.

We comment on the use of a sparse storage scheme for the explicit solution. The compact MSR implementation is significantly more compact than other sparse schemes; see Table 3.1 on Page 44. It typically provides faster solution times over other schemes due to better caching and high memory I/O. The time results for the modified offset-labelled MTBDDs are, in fact, comparable with the explicit methods based on other storage schemes; see e.g. sparse results for these case studies listed on the PRISM web page [Pri], or [Par02].

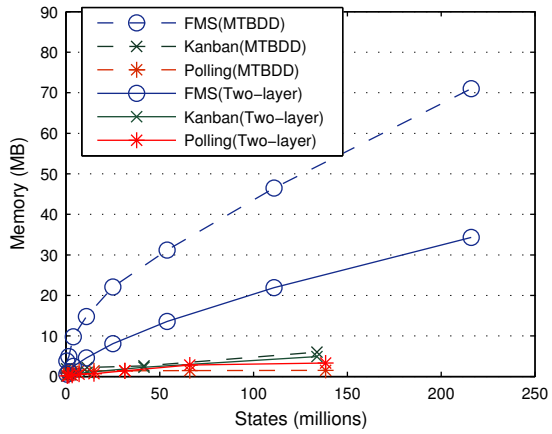
5.4.3 Memory Consumption

In this section, we consider memory requirements in more detail. For both the original and the modified offset-labelled MTBDDs, the memory requirements consist of the matrix, the diagonal vector, the iteration vector (n doubles), and the vector $\tilde{X}$ (see Section 3.1 and 4.2). The storage for iteration vector is, in fact, independent of the approach used, and is thus not considered in our analysis. We also ignore here the storage for the diagonal vector, although the original offset-labelled MTBDD uses 8 bytes to store each entry of the diagonal vector while the modified data structure uses 2 bytes. Obviously, the memory for the matrix depends heavily on the data structure used. The vector $\tilde{X}$ is also affected since its size n_{max} is equal to the largest matrix block used, which is governed by the choice of l_b. To explain this more clearly, in Figure 5.9, we plot memory usage against number of states. Figure 5.9(a) shows memory for the matrix alone, and Figure 5.9(b) shows memory for the matrix and $\tilde{X}$ vector combined.

Our first observation, in Figure 5.9(a), is that the increase in memory for the modified data structure (Two-layer) over the original data structure (MTBDD) is reasonably small. The increase in explicit storage due to the condition $l_b + l_{sm} = l_{total}$ is balanced by the use of the more efficient compact MSR format. Furthermore, when we consider the memory for matrix and vector combined in Figure 5.9(b), we see that the modified data structure actually requires less memory for large CTMCs. This is because we are able to select a larger value of l_b, making the maximum block size smaller, i.e. re-



(a) Off-diagonal Matrix



(b) Off-diagonal Matrix and vector

Figure 5.9: Memory usage plotted against number of states

ducing n_{max}. For FMS ($k = 13$), for example, the total memory requirements (matrix and vector) for the modified and the original MTBDDs are 35MB and 71MB, respectively. It is interesting to mention here that the standard MSR format for this FMS model requires nearly 27GB to store the CTMC matrix.

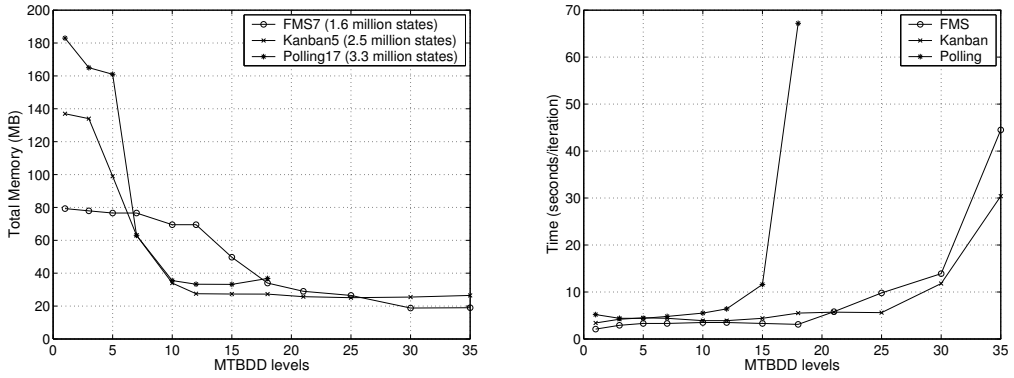
5.5 Experimental Results: Out-of-core

We have developed an out-of-core implementation, where the modified offset-labelled MTBDD storage of the matrix is kept in RAM, as before, but (iteration and diagonal) vectors are stored on disk and retrieved as required. This idea has been described previously in Chapter 4, where an out-of-core version of the original offset-labelled MTBDD-based numerical solution was implemented in similar fashion. In this section, we analyse this modified symbolic out-of-core implementation. We use the same case studies as in the previous section. These results show that using out-of-core instead of in-core storage for the vectors allows significantly larger models to be solved on a standard workstation. The largest model we solved using the in-core implementation was the Polling system with 31 million states. Taking out-of-core approach, we successfully solve a CTMC with over 384 million states.

In Section 5.5.1, we analyse the memory and time properties of the out-of-core implementation against a range of values for the number of block levels l_b. From this analysis, as before, we derive a heuristic for a priori selection of this parameter. We present the numerical solution times for the modified data structure in Section 5.5.2. These results are compared with the original offset-labelled MTBDDs and an explicit sparse out-of-core implementation. To demonstrate the scalability of the out-of-core method, in Section 5.5.3, we present results of the out-of-core implementation on a more powerful workstation. Finally, in Section 5.5.4, we propose some future work to improve the performance of the symbolic out-of-core method.

5.5.1 A Good Choice for the Parameter l_b

An MTBDD comprises a total number of levels l_{total}. The modified offset-labelled MTBDD data structure comprises two layers of symbolic sparse



(a) Total Memory (b) Time per iteration

Figure 5.10: Timing results: A comparison of out-of-core methods

storage. The size of the top layer of the data structure is governed by the parameter l_b, and the bottom layer is determined by the parameter l_{sm}, with an additional condition $l_b + l_{sm} = l_{total}$. This condition is asserted to implement the standard Gauss-Seidel iterative method. We know that selection of this parameter l_b determines l_{sm} and controls the size of the top and bottom layers of the two offset-labelled MTBDD data structures. For the out-of-core solution, the parameter l_b also influences the amount of disk I/O the process has to perform. This parameter, hence, plays a crucial role in obtaining performance from the out-of-core solution. A heuristic to a priori select the parameter l_b for the in-core solution based on the modified data structure was derived in Section 5.4.1. In this section, we seek a heuristic to select l_b for the out-of-core solution. As before, we search for a heuristic which provides good performance, although determining the optimal value for l_b is likely to be expensive or infeasible.

In Figure 5.10, we illustrate the total in-core memory requirement and time per iteration for the symbolic out-of-core solution based on the modified offset-labelled MTBDD CTMC storage. The figure plots these results against a range of values for the number of top-layer levels l_b, for the three CTMCs, one from each case study. The total in-core memory includes storage for the off-diagonal matrix, and for the diagonal and iteration vector blocks.

We first explain the memory characteristics of the out-of-core solution, shown in Figure 5.10(a). Our first observation in the figure is that the three

CTMCs depict similar patterns. As expected, increasing l_b decreases the amount of total memory required by the process. There is an increase in the total memory as the parameter l_b approaches l_{total} (for the values of l_{total}, see Table 5.1 on Page 102; see also Figure 5.6). This behaviour, however, is not illustrated in Figure 5.10(a) because the memory properties are plotted for the maximum value $l_b = 35$, for the reasons which will be apparent in the next paragraph.

The time per iteration properties for the out-of-core implementation are plotted in Figure 5.10(b) against the parameter l_b. The three CTMCs show similar behaviour, i.e. the time per iteration increases as the number of block levels l_b for a CTMC approaches (the values close to) its respective number of total levels, l_{total}. The reasons are two-fold. Firstly, increasing l_b (for the values close to l_{total}) may cause an increase in the memory required to store the CTMC; see Figure 5.6. Secondly, an increase in l_b increases the amount of disk I/O the process has to perform because the iteration vector is read 2^{l_b} times from disk. The solution times for the values of l_b close to the respective l_{total} were relatively very large. For these reasons, in Figure 5.10, we have not plotted the memory and time properties for larger values (close to l_{total}) of l_b.

We conclude this section as follows. For the in-core implementation of the modified offset-labelled MTBDD data structure, we adopted the heuristic $l_b = 0.6 \times l_{total}$. For the out-of-core implementation, however, this heuristic results in a high amount of disk I/O. A low value for the parameter l_b cannot be selected because the resulting data structure will require large memory for the bottom layer storage. We select $l_b = 0.4 \times l_{total}$ to compromise between memory and time. This heuristic assumes that sufficient memory is available to store all of the required data structures. The solution times reported in the next sections are collected using this heuristic. However, for larger models, we manually select values for the parameter l_b if the heuristic results in a memory which is larger than the available RAM. A more involved heuristic can also be devised which takes into consideration the amount of RAM allocated to the solution process (or the RAM available in a workstation).

Model	k	States (n)	a/n	Time (sec. per iter.)			Iterations	
				Sparse	MTBDD	Two-layer	PGS	GS
FMS	6	537,768	7.8	1.1	1.3	1.0	1027	812
	7	1,639,440	8.3	3.8	3.2	3.4	1207	966
	8	4,459,455	8.6	10.7	10.4	9.6	1392	1125
	9	11,058,190	8.9	51.8	35.9	26.2	1581	1287
	10	25,397,658	9.2	146	142	182	1775	1454
	11	54,682,992	9.5	374	708	788	1972	1624
	12	111,414,940	9.7	–	1,554	2319	2174	1798
	13	216 427 680	9.9	–	3,428	6086	2379	1977
Kanban	4	454,475	8.8	1.0	1.3	0.6	414	323
system	5	2,546,432	9.6	6.0	4.5	4.0	590	461
	6	11,261,376	10.3	68.6	22.6	21	794	622
	7	41,644,800	10.8	283	143	120	1022	802
	8	133,865,325	11.3	–	601	558	1151	999
	9	384,392,800	11.6	–	–	2049	–	1211
Polling	15	737,280	8.3	1.2	0.8	0.9	179	32
system	16	1,572,864	8.8	2.9	2.0	2.1	196	33
	17	3,342,336	9.3	6.4	4.6	4.6	213	34
	18	7,077,888	9.8	20.4	10.5	10.7	229	34
	19	14,942,208	10.3	71	23.8	25.1	255	35
	20	31,457,280	10.8	162	52	79.1	271	36
	21	66,060,288	11.3	359	177	212	340	36
	22	138,412,032	11.8	–	374	449	353	37
	23	289,406,976	12.3	–	–	1518	–	38

Table 5.3: Comparing solution times for the out-of-core methods

5.5.2 Speed of the Numerical Solution

We now present timing results for the modified symbolic out-of-core method. We compare this method with the symbolic out-of-core method of Chapter 4 which stores CTMCs using original offset-labelled MTBDDs, and with the complete out-of-core method of Chapter 3. The three methods share the fact that they all provide remedy for both the matrix and vector storage. Table 5.3 reports these results. Results for the solution based on the original and the modified MTBDDs are listed under "MTBDD" and "Two-layer" respectively. Results for the complete out-of-core method are reported under "Sparse". In each case, we use the most efficient numerical solution method available. The iterative methods were tested for the convergence criterion given by Equation (2.16) for $\varepsilon = 10^{-6}$.

The first four columns in Table 5.3 report the model statistics. Columns $5-7$ list times per iteration for the three methods. The number of iterations

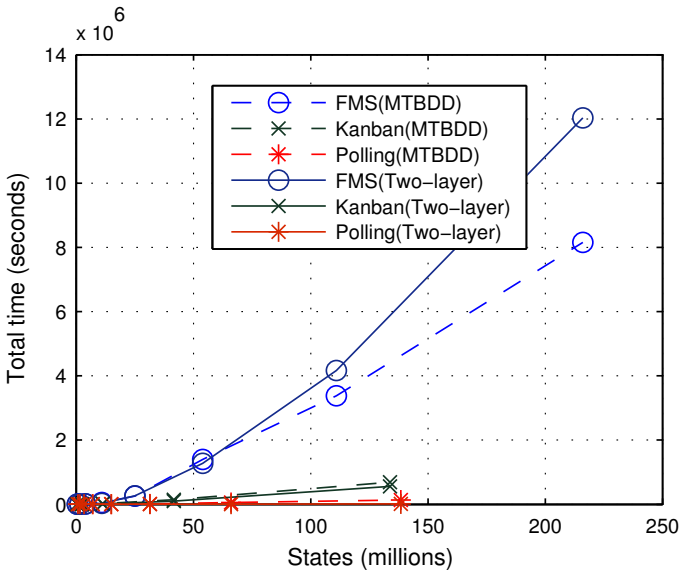


Figure 5.11: Out-of-core total times plotted against number of states

for the methods are reported in column $8-9$. The symbolic method based on the original offset-labelled MTBDDs uses the pseudo Gauss-Seidel (PGS), the other two out-of-core methods use the Gauss-Seidel method. Total times for the solution can be calculated using the times per iteration and the number of iterations.

Comparing with the complete out-of-core method, we see in Table 5.3, that the modified offset-labelled MTBDD times for the Kanban and Polling systems are typically twice as fast as the explicit method. Note also that the modified approach can handle larger CTMCs than the explicit method. This is due to the fact that the explicit out-of-core approach is limited by the disk memory available with a workstation (in this case, 6GB). Finally, we note that the explicit method provides relatively faster solution times for the FMS CTMCs. We are already aware of the reasons for this: that FMS is relatively less structured, requires larger MTBDD for its storage, and therefore requires relatively more work from a symbolic method. The explicit method, on the other hand, performs consistently across the three models.

We now concentrate on comparing the two symbolic methods. As for the in-core implementations, to give a better indication of the trends for the

two methods in Table 5.3, we plot total solution times against the number of states in Figure 5.11. As in Figure 5.8, dashed lines are used to draw the results for the original offset-labelled MTBDDs, and are tagged with "MTBDD", while the results for modified offset-labelled MTBDDs are drawn in solid lines and are tagged with "Two-layer".

Our first observation in Figure 5.11 is that the modified symbolic method provides faster solutions for Polling and Kanban systems, i.e. for these two models, the dashed lines are above the solid lines. For FMS system, for model sizes of up to 50 million states, the modified offset-labelled MTBDD data structure provides faster speeds than the original offset-labelled MTBDDs. The performance however declines afterwards (see the plots). This is explained as follows; see also Section 5.5.1. For the out-of-core solution (modified offset-labelled MTBDD), selecting a larger value for l_b increases the disk I/O and consequently imposes a time overhead on the solution process. Unfortunately, selecting a lower value for l_b has an associated cost in that it results in an increased storage for the bottom layer of the data structure. This amount of resulting memory may not be available with a workstation. This effect is not as significant for structured models and hence Polling and Kanban systems perform well. For FMS, which is less structured and requires a larger MTBDD, the in-core memory requirement of the solution process (for larger models) approaches the RAM limits of the workstation. Consequently, it slows down the solution speed. To investigate this further, in Section 5.5.3, we present results from the out-of-core implementation of the modified symbolic out-of-core method on a workstation with larger RAM.

5.5.3 Scalability of the Out-of-Core Solution

In Table 5.4, we present timing results for the symbolic out-of-core solution (modified offset-labelled MTBDDs) on a more powerful workstation, an Intel Pentium 4, 2.80GHz CPU machine running Linux with 512 KB cache, 1GB RAM, and a 60GB SCSI local disk. As before, the first four columns in the table list the model statistics. The next three columns list the time per iteration, total time and the number of iterations. The purpose of presenting these results is to demonstrate the scalability of the symbolic out-of-core solution, and to further investigate its behaviour. We use machine2 to refer to this faster workstation and use machine1 to refer to the workstation used

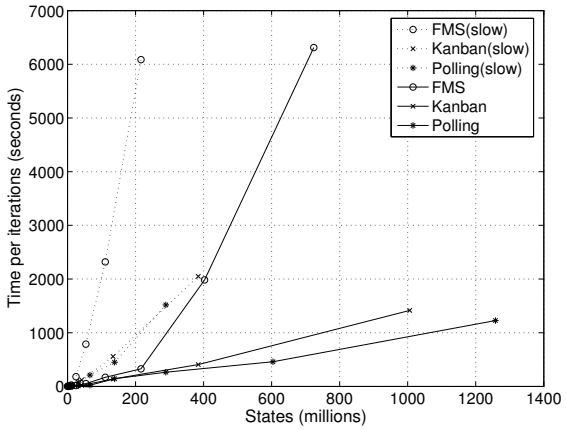
Model	k	States (n)	a/n	Times		Iterations
				Per iteration (seconds)	Total (hr:min:sec)	
FMS	6	537,768	7.8	0.18	2:27	812
	7	1,639,440	8.3	0.75	12:05	966
	8	4,459,455	8.6	1.6	30:00	1125
	9	11,058,190	8.9	7.4	2:38:44	1287
	10	25,397,658	9.2	17.4	7:01:40	1454
	11	54,682,992	9.5	51.6	23:16:39	1624
	12	111,414,940	9.7	170	84:54:20	1798
	13	216 427 680	9.9	327	179:34:39	1977
	14	403,259,040	10.03	1984	–	> 50
	15	724,284,864	10.18	6312	–	> 50
Kanban system	4	454,475	8.8	0.1	33	323
	5	2,546,432	9.6	0.8	6:09	461
	6	11,261,376	10.3	5.0	51:50	622
	7	41,644,800	10.8	18.9	4:12:38	802
	8	133,865,325	11.3	139	38:34:21	999
	9	384,392,800	11.6	407	136:54:37	1211
	10	1,005,927,208	11.97	1424	566:49:52	1433
Polling system	15	737,280	8.3	0.1	4	32
	16	1,572,864	8.8	0.3	10	33
	17	3,342,336	9.3	0.9	31	34
	18	7,077,888	9.8	2.1	1:12	34
	19	14,942,208	10.3	4.6	2:41	35
	20	31,457,280	10.8	10.1	6:04	36
	21	66,060,288	11.3	22.8	13:41	36
	22	138,412,032	11.8	143	1:28:11	37
	23	289,406,976	12.3	264	2:47:12	38
	24	603,979,776	12.8	460	4:51:20	38
	25	1,258,291,200	13.3	1226	13.16:54	39

Table 5.4: Improved Symbolic out-of-core solution method on machine2

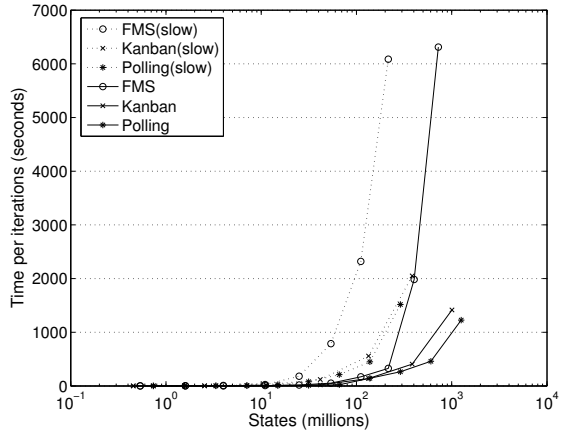
earlier in this chapter (440MHz, 512MB RAM, 6GB disk).

Our first observation in Table 5.4 is that it is now possible to solve even larger models. We successfully solve a Kanban system with over 1 billion states (in under 24 days), and a Polling system with over 1.2 billion states (in under 14 hours). Note also that it is now feasible to solve even larger FMS systems (724 million states). Our second observation is that the computations can be performed much faster. For example the solution of the FMS system ($k = 13$) which would have taken over 139 days (see Table 5.3) on machine1, took under 8 days to converge on machine2.

The above results are, of course, to be expected since we are now running on a more powerful hardware. However, we also make the following important



(a) Linear scale



(b) Logarithmic scale

Figure 5.12: Comparing out-of-core times for the two machines

observation. Comparing the results for machine2 (Table 5.4) with the results for machine1 (Table 5.3), we note that the out-of-core times for larger models on machine2 are on average 10 to 20 times faster than the times on machine1, although the CPU speeds' ratio for the two workstations is only 7 (2800/440). See, for instance, the results for the FMS system ($k = 13$) which demonstrate a more than 18-fold increase in solution speed. To investigate this further, we plot timing results for the two workstations in Figure 5.12. The times per iteration for the three models are plotted against the number of states. The plots with dashed lines are drawn for machine1 and these are marked "slow". To improve visibility, the plots are drawn in both linear and logarithmic scales. For smaller models, in all three case studies, we observe a linear increase in time per iteration against the number of states. However, as the models grow larger, near the end of the plots (towards the right), we note a steep increase in time for all three case studies. The increase in times is particularly rapid for the FMS system. The two machines show similar patterns, albeit with some offset in the number of states, i.e. the patterns in the plots are similar but these are shifted towards right for machine2. These patterns are more obvious for the plots in the logarithmic scale. These findings are explained as follows.

For an increase in the model sizes, there is a proportional increase in the computational and space requirements of the solution. This increase in the process requirement is typically linear. However, as the memory required by the process approaches the RAM limits of the workstation, the respective increase in solution time is no longer linear in the model size (number of states). For these reasons, using a workstation with larger RAM results in relatively faster speeds for the solution of larger models.

For the symbolic out-of-core method, the required memory can be decreased by selecting a larger value for l_b. This does not help however, because increasing l_b causes an increase in the amount of disk I/O (see also Section 5.5.1). Note that increasing l_b also increases storage required for the top layer, albeit to a lesser extent. These observations are equally valid for all three case studies. The FMS model is less structured and therefore is affected the most. We conclude here with a final note that the symbolic out-of-core solution based on modified offset-labelled MTBDD CTMC storage scales well to more powerful workstations.

5.5.4 Proposed Improvements

In this section we propose some modifications to the symbolic out-of-core method which will possibly improve its performance. We know that the modified offset-labelled MTBDD data structure has two parameters, l_b and l_{sm}, where $l_b + l_{sm} = l_{total}$. Selecting a value for l_b also determines l_{sm}. The parameter l_b governs the amount of memory required for the top and bottom layers of the storage, as well as the amount of disk I/O. It also determines n_{max}, the size of the largest vector block to be kept in RAM. In Tables 5.3 and 5.4, we note that for larger models, the speed of the solution is relatively poor compared to the speed for the smaller models. Note the speeds for the FMS models in particular. This is because, in these cases, the selected value of l_b requires a large amount of disk I/O and hence significantly decreases the solution speed. A smaller value for l_b cannot be selected because it increases l_{sm}, which consequently increases the storage requirements for CTMC matrix above the amount of the available RAM (see Section 5.5.1).

We note that this unfortunate situation arises because the storage requirements and the disk I/O are dependent on each other, i.e. a decrease in the storage for the solution process causes a decrease in its speed and vice versa. In fact, it is possible to separate the two issues, i.e. to separate disk I/O from the parameter l_b. Let us define an additional parameter l_d, such that $l_d < l_b$. This parameter l_d will be used to decompose the vector into blocks and therefore will now govern the value of n_{max}, as well as the amount of disk I/O. The parameter l_b determines the top-layer decomposition of the CTMC matrix and hence will only control the size of storage for the top layer of the data structure.

We believe that these modifications to the symbolic out-of-core method will result in substantial improvements in its performance. Note that the parameter l_d controls n_{max}, and therefore selecting smaller values for l_d will result in larger values for n_{max}.

CHAPTER

SIX

Conclusions

Our aim in this thesis was to develop efficient techniques for the steady-state analysis of large continuous time Markov chains on a single, contemporary workstation. Earlier work in this context has focused on implicit and parallel explicit solutions. We considered the out-of-core approach and applied it to both implicit and explicit methods.

6.1 Epitome and Assessment

The state space explosion problem has been a major hurdle in the progress of the steady-state analysis of large CTMCs. Much research is focused on the development of methods and data structures which can minimise the space and time requirements of this process. During the last decade, some progress has been made, beginning with the solution of a 15 million state system on a workstation with 128MB RAM, with no assumption on its structure [DS97, DS98a]. This approach was parallelised in 1999 [KH99, Kno99], leading to the solution of a CTMC with 724 million states on a 26-node dual-processor cluster [BH01]. Another direction taken to combat the state space explosion problem has been the development of implicit methods which rely on exploiting structure and regularity in the CTMCs. These included Kronecker methods [Pla85, Don94, Kem96, CT96, BCDK97, FPS98], Matrix Diagrams [CM99, Min00, Min01], and the offset-labelled MTBDDs [Par02, KNP04b]. 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, even for parallel and distributed techniques.

In Chapter 3, we introduced the complete out-of-core solution method. The method relaxed the size limitation on the explicit solution methods, caused by the need for the in-core, explicit storage of the probability vector. It makes no assumptions about the structure of the matrix, and hence can be applied to CTMCs in general. The method employed the compact MSR sparse storage format which itself was a contribution of this work. The scheme offers a 30% memory saving over current sparse storage schemes, and it can be used with in-core, out-of-core or with distributed iterative numerical solutions of CTMCs.

In Chapter 4, we introduced the symbolic out-of-core method. Two implementations of the symbolic method were described and compared using experimental results. The symbolic method also relaxed the limitations for the implicit methods, which was caused by the need to store the probability vector in-core, explicitly. It was demonstrated that the symbolic method can be used to solve models as large as 216 million states on a workstation with 512MB RAM. Even larger models can be solved by decomposing the vector into a larger number of blocks, i.e. by reducing the in-core memory requirement for the solution process. Since equivalent implicit approaches require an iteration vector of size proportional to the state space, the largest model these techniques can solve on equivalent hardware is of size 60 million states. We analysed the symbolic out-of-core method in detail and presented a speed comparison with the explicit methods. Notes on improving performance of the solution method were also outlined in Section 4.3.1.

In Chapter 5, we presented modifications to the offset-labelled MTBDD data structure and addressed one of its main deficiencies by presenting an efficient implementation of the Gauss-Seidel method. We gave experimental results from its implementation, compared them to existing MTBDD-based implementations and showed that the modified data structure improves the time and memory properties of its predecessor. We also presented results from an out-of-core version of the modified offset-labelled MTBDDs, and demonstrated solutions of very large CTMCs. Finally, we gave suggestions for how to improve the performance of the out-of-core solution method.

In summary, the aim of this thesis was to extend the size of the models which can be analysed on a contemporary workstation. The techniques presented in this thesis, we believe, have successfully achieved this aim. We used a workstation of modest specifications for our implementations and experimental analysis. We demonstrated solutions of models with up to 384 millions states and 4 billion transitions on this workstation. Furthermore, on a more powerful workstation, we analysed models as large as 1.2 billion states and 16 billion transitions. Using any of the existing alternative techniques, analysis of models of such sizes on a single contemporary workstation is not possible.

Secondly, work on the integration of our developed techniques into the tool PRISM is in process. The improved offset-labelled MTBDD and compact MSR data structures have been integrated into the official development branch of PRISM and will be included in the next public release. The out-of-core solution methods will be integrated into PRISM in the future.

6.2 Future Research

We envisage a number of possibilities for future development of the work presented in this thesis. Two major directions can be taken in this context. Firstly, work can be carried out to improve the performance of the techniques introduced in this thesis. Secondly, these techniques can be applied to other domains, e.g. the techniques can be augmented by their parallelisation or can be extended to other modelling formalisms.

We first discuss ideas for how to improve the performance of our techniques. In Sections 4.3.1 and 5.5.4, we have already presented some suggestions to enhance the performance of the symbolic out-of-core method. However, in general, the out-of-core algorithms presented in this thesis suffer 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 performing its computation. 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.

Another area for improvements lies in the optimisations for the out-of-core implementations at the level of the operating system. The out-of-core

algorithms may be implemented with two threads instead of two processes. This can enhance solution speeds because threads share a single address space and communication at this level is typically faster. Secondly, mapping the disk memory onto the process address space (memory-mapped I/O) will also increase the solution speed because the standard I/O requires additional time to copy data buffers from the kernel space to the user space. Thirdly, using facilities at the kernel level of the operating system (e.g using a larger size for disk blocks) to ensure lower time for disk *seek* operations will also improve the solution speed for the out-of-core methods. Finally, redundant arrays of independent disks (RAID) can also be used to increase the disk throughput. Analogous to the distributed computers, which mainly provide compute power and larger memories, RAIDs deliver high disk throughput by connecting disks in parallel.

We now consider applications of our techniques to other domains in the performance analysis. A first possibility in this direction exists in generalisation of our storage and out-of-core techniques to other numerical computation problems, such as the transient analysis of CTMCs and the analysis of both DTMCs and MDPs. The computations involved in such analyses are also based on matrix-vector product (MVP) operation, which is the core operation for the steady-state CTMC analysis. Secondly, the out-of-core techniques, both explicit and symbolic, can be used to implement Krylov subspace methods which are also based on multiple MVP operations. We know from Section 2.4 that Krylov methods provide typically faster convergence but are limited because they require storage for a large number of iteration vectors.

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 (in-core) storage of the solution vector. The idea of a symbolic out-of-core technique is a promising one, and is equally applicable to Kronecker methods, i.e. the out-of-core storage of the iteration vector can provide an improvement.

Probably the most important extension of our work is its integration with parallel and distributed approaches. In this context, we first consider the complete out-of-core method of Chapter 3 which is the most generally applicable technique (because it does not assume structure in the models).

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 distributed architecture 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. A distributed implementation of the complete out-of-core method will enable very large models to be solved on clusters of modest sizes by reducing the main memory (RAM) requirements. Furthermore, we believe that it will improve the overall performance for the distributed out-of-core methods because this approach provides computational work to overlap with the idle time during interprocessor communications.

The improved offset-labelled MTBDD data structure introduced in Chapter 5 provides a direct access to blocks of a CTMC matrix, as well as to its individual matrix rows. Essentially, this combines, in a single data structure, both the compactness of the symbolic representations and the flexibility of sparse schemes. The data structure, hence, constitutes an ideal basis for a distributed implementation. Initial work in this direction [KPZM04] which reports a dual-processor shared memory implementation of the improved offset-labelled MTBDD data structure is promising. Further work can be carried out to extend this approach to the fully distributed case. Based on this data structure, a distributed implementation of the symbolic out-of-core method will further extend the size of the models which can be analysed.

6.3 Conclusion

We set out out to develop out-of-core techniques for the steady-state solution of large CTMCs as a means to extend the size of solvable models on a single workstation. We successfully achieve this aim and are able to solve models over an order of magnitude larger than possible previously. Since the techniques, both symbolic and explicit, presented in this thesis are suitable for block based methods, these can be easily augmented with parallel and distributed techniques.

The so-called implicit and structural (or modular) decomposition approaches have also shown significant developments, particularly, in the past few years. We have demonstrated the effectiveness of such an approach

(offset-labelled MTBDDs) in this thesis.

An increasing trend in the area of distributed and grid computing has been observed. Major developments have been made in the area of communication technologies, both in software and hardware. Disk technology has also benefited from the introduction of RAIDs.

In the future, we anticipate that a combination of parallel and out-of-core techniques will play an important role in the analysis of large CTMCs.

APPENDIX

A

The PRISM Tool

In this appendix, we give a brief overview of the PRISM tool [KNP02a, KNP04a]. PRISM, the **Pr**obabilisti**c** **S**ymbolic **M**odel **C**hecker, is a tool for analysing probabilistic systems. It supports three types of models: discrete time Markov chains, continuous time Markov chains and Markov decision processes. BDDs and MTBDDs are the basic underlying data structures of PRISM. For the numerical solution phase, however, it provides three engines: one using conventional MTBDDs, one based on in-core sparse iterative methods, and a third which uses offset-labelled MTBDDs. Further information about the PRISM tool can be obtained from the PRISM website (http://www.cs.bham.ac.uk/~dxp/prism/), including the user guide, source code of the tool, relevant papers and details of over thirty case studies.

APPENDIX

B

Case Studies

We have benchmarked implementations of the techniques presented in this thesis using three CTMC case studies. These are: the flexible manufacturing system of [CT93], the Kanban manufacturing system of [CT96] and the cyclic server polling system of [IT90]. We often abbreviate the names of these case studies to "FMS", "Kanban" and "Polling", respectively. These three examples have been widely used as benchmarks, both for explicit and implicit techniques, see for example [Kno99, BH01, DS98a, CM99, Cia01b, KNP04b].

The case studies have been generated using version 1.3.1 of the PRISM tool [KNP04a]. We are grateful to Gethin Norman for developing these case studies for our benchmarks. We give a brief description of the three case studies in the following sections.

B.1 Cyclic Server Polling System

The cyclic server polling system, a CTMC model, was presented by Ibe and Trivedi in [IT90]. The Polling system consists of k stations or queues and a server. The server polls the stations in a cycle to determine if there are any jobs in the station for processing. The number of states in the CTMC increases with an increase in the number of stations.

B.2 Kanban Manufacturing System

Ciardo and Tilgner presented a CTMC model, the Kanban manufacturing system, in [CT96], and used it to benchmark their Kronecker-based solution of CTMCs. The Kanban model comprises four machines. The model parameter k represents the maximum number of jobs that may be in a machine at one time.

B.3 Flexible Manufacturing System (FMS)

In [CT93], Ciardo and Tilgner presented the flexible manufacturing system (FMS) model to benchmark their decomposition approach for the solution of large stochastic reward nets (SRNs), a class of Markovian stochastic Petri nets [Mol82]. The FMS model comprises three machines which process different types of parts. One of the machines may also be used to assemble two parts into a new part. The total number of parts in the system is kept constant. The model parameter k denotes the maximum number of parts which each machine can handle.

BIBLIOGRAPHY

[ADK97] S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In *Proceedings of the Parallel Computing Conference (PARCO '97)*, pages 581–588, 1997.

[AKH97] S. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In *Proc. PNPM'97*, pages 112–121. IEEE Computer Society Press, 1997.

[Axe96] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1996.

[Bau93] F. Bause. Queueing Petri Nets: A Formalism for the Combined Qualitative and Quantitative Analysis of Systems. In *Proc. PNPM'93*, pages 14–23. 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.

[BCDK97] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Kronecker Operations and Sparse Matrices with Applications to the Solution of Markov Models. ICASE Report 97-66, Institute for

Computer Applications in Science and Engineering, December 1997.

[BCDK00] P. Buchholz, G. Ciardo, Susanna Donatelli, and P. Kemper. Complexity of memory-efficient kronecker operations with applications to the solution of markov models. *INFORMS J. on Computing*, 12(3):203–222, 2000.

[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):248–260, 1975.

[BDKW03] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based Parallel Computation of Passage Time Densities in Large Semi-Markov Models. In *Proc. 4th International Meeting on the Numerical Solution of Markov Chains (NSMC 2003)*, pages 99–120, Chicago, USA, 2003.

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

[Bel03] A. Bell. *Distributed Evaluation of Stochastic Petri nets*. PhD thesis, RWTH Aachen, 2003.

[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.

[BFK99] P. Buchholz, M. Fischer, and P. Kemper. Distributed Steady State Analysis Using Kronecker Algebra. In *Proc. NSMC'99*, pages 76–95, 1999.

[BG96] M. Bernardo and R. Gorrieri. Extended Markovian Process Algebra. In *Proc. of the 7th Int. Conf. on Concurrency Theory (CONCUR 1996), Pisa, Italy*, volume 1119 of *LNCS*, pages 315–330. Springer-Verlag, 1996.

[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.

[BK01] P. Buchholz and P. Kemper. Compact Representations of Probability Distributions in the Analysis of Superposed GSPNs. In Reinhard German and Boudewijn Haverkort, editors, *Proc. PNPM'01*, pages 81–90, September 2001.

[BK04] P. Buchholz and P. Kemper. Kronecker based Matrix Representations for Large Markov Models. In *Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 256–295. Springer-Verlag, 2004.

[BM99] M. Bozga and O. Maler. On the Representation of Probabilities over Structured Domains. In N. Halbwachs and D. Peled, editors, *Proceedings of CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 261–273. Springer-Verlag, July 1999.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Buc99a] P. Buchholz. Projection Methods for the Analysis of Stochastic Automata Networks. In *Proc. NSMC'99*, pages 149–168, 1999.

[Buc99b] P. Buchholz. Structured analysis approaches for large Markov chains. *Appl. Numer. Math.*, 31(4):375–404, 1999.

[Buc00] P. Buchholz. Multilevel Solutions for Structured Markov Chains. *SIAM Journal on Matrix Analysis and Applications*, 22(2):342–357, 2000.

[CCM95] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. De Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935

of *Lecture Notes in Computer Science*, pages 181–200. Springer-Verlag, 1995.

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

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

[CGN98] G. Ciardo, J. Gluckman, and D. Nicol. Distributed State-Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 10(1):82–93, 1998.

[Cia01a] 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, LNCS 2090*, pages 344–374. Springer-Verlag New York, Inc., 2001.

[Cia01b] 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 2001.

[CM99] G. Ciardo and A.S. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proc. PNPM'99*, pages 22–31, Zaragoza, 1999.

[CSG99] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware Software Approach*. Morgan Kaufmann Publishers, 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.

[CT97] G. Ciardo and M. Tilgner. Parametric State Space Structuring. ICASE Report 97-67, Institute for Computer Applications in Science and Engineering, December 1997.

[Don94] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In R. Valette, editor, *Proc. 15th International Conference on Application and Theory of Petri Nets (APN'94)*, volume 815 of *LNCS*, pages 258–277. Springer, 1994.

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

[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.

[DS00] T. Dayar and W.J. Stewart. Comparison of Partitioning Techniques for Two-Level Iterative Solvers on Large, Sparse Markov Chains. *SIAM Journal on Scientific Computing*, 21(5):1691–1705, 2000.

[DV99] I.S. Duff and H.A. van der Vorst. Developments and trends in the parallel solution of linear systems. *Parallel Computing*, 25(13–14):1931–1970, 1999.

[FPS98] P. Fernandes, B. Plateau, and W.J. Stewart. Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, 1998.

[GL96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. *Lecture Notes in Computer Science*, 2057:217–234, 2001.

[GO93] G.H. Golub and J.M. Ortega. *Scientific Computing: an Introduction with Parallel Computing*. Academic Press, 1993.

[HBB99] B.R. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proc. NSMC'99*, pages 12–21. IEEE Computer Society Press, 1999.

[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.

[HKN+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for Performability Analysis and Verification of Stochastic Systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, pages 23–67, 2003.

[HMKS99] 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)*, pages 188–207, Zaragoza, 1999.

[HMPS96] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian Analysis of Large Finite State Machines. *IEEE Transactions on CAD*, 15(12):1479–1493, 1996.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[HR94] H. Hermanns and M. Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In *Proc. of 2nd Workshop on Process Algebras and Performance Modelling, Regensberg, Germany*, pages 71–88, 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.

[Kem96] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(9):615–628, 1996.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cumming Publishing Company, 1994.

[KH99] W.J. Knottenbelt and P.G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In *Proc. Numerical Solution of Markov Chains (NSMC'99)*, pages 58–75, Prensas Univerversitarias de Zaragoza, 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*, pages 135–151, July 2002. Available as Volume 2399 of *LNCS*.

[KMHK98] W.J. Knottenbelt, M. Mestern, P.G. Harrison, and P. Kritzinger. Probability, parallelism and the state space exploration problem. In R. Puigjaner, N.N. Savino, and B. Serra, editors, *Computer Performance Evaluation*, volume 1469 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 1998.

[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).

[Kno99] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, UK, February 1999.

[KNP02a] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. TOOLS'02*, volume 2324 of *LNCS*, pages 200–204, April 2002.

[KNP02b] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *Proc. TACAS 2002*, pages 52–66, April 2002. Available as Volume 2280 of *LNCS*.

[KNP04a] M. Kwiatkowska, G. Norman, and D. Parker. Prism 2.0: A tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323, 2004.

[KNP04b] 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)*, 6(2):128–142, 2004.

[KPZM04] M. Kwiatkowska, D. Parker, Y. Zhang, and R. Mehmood. Dual-processor parallelisation of symbolic probabilistic model checking. In *Proc. 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 123–130, 2004.

[KSW04] M. Kuntz, M. Siegle, and E. Werner. Symbolic performance and dependability evaluation with the tool CASPA. In *Proc. 1st European Performance Engineering Workshop (EPEW'04), FORTE'04 workshop*, volume 3236 of *LNCS*, pages 293–307. Springer, 2004.

[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):93–122, 1984.

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

[MCC97] P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models: A Distributed Solution Tool. In *Proc. PNPM'97*, pages 122–131, 1997.

[Meh03] R. Mehmood. A Survey of Out-of-Core Analysis Techniques in Stochastic Modelling. Technical Report CSR-03-7, School of Computer Science, University of Birmingham, UK, 2003.

[Meh04] R. Mehmood. Serial Disk-based Analysis of Large Stochatic Models. In *Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2004.

[Min00] A.S. Miner. *Data Structures for the Analysis of Large Structured Markov Chains*. PhD thesis, Department of Computer Science, College of William & Mary, Virginia, 2000.

[Min01] A.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.

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

[MP04] A.S. Miner and D. Parker. Symbolic Representations and Analysis of Large Probabilistic Systems. In *Validation of Stochastic*

Systems: A Guide to Current Research, volume 2925 of *Lecture Notes in Computer Science*, pages 296–338. Springer-Verlag, 2004.

[MPK03a] R. Mehmood, D. Parker, and M. Kwiatkowska. An Efficient Symbolic Out-of-Core Solution Method for Markov Models. Technical Report CSR-03-8, School of Computer Science, University of Birmingham, UK, August 2003.

[MPK03b] 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, UK, December 2003.

[MPS99] V. Migallon, J. Penades, and D. Szyld. Block two-stage methods for singular systems and Markov chains. In *Proc. Numerical Solution of Markov Chains (NSMC'99)*, Prensas Univerversitarias de Zaragoza, 1999.

[PA91] B. Plateau and K. Atif. Stochastic Automata Network for Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 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*, pages 147–153, 1985.

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

[PSS92] B. Philippe, Y. Saad, and W.J. Stewart. Numerical methods in Markov chain modeling. *Operations Research*, 40(6):1156–1179, 1992.

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

[Saa91] Y. Saad. Projection methods for the numerical solution of Markov Chain Models. In W.J. Stewart, editor, *Proc. of the 1st International Workshop on the Numerical Solution of Markov Chains*, pages 455–472, Raleigh, NC, USA, 1991. Marcel Dekker Inc., NJ.

[Saa95] Y. Saad. Preconditioned Krylov subspace methods for the numerical solution of Markov chains. In W.J. Stewart, editor, *Computations with Markov Chains: Proc. of the 2nd International Workshop on the Numerical Solution of Markov Chains*, pages 49–64, Raleigh, NC, USA, 1995. Kluwer Acaedmic Publishers.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Second edition, 2003.

[Sie01] M. Siegle. Advances in Model Representation. 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.

[Son89] P. Sonneveld. CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52, January 1989.

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

[SV00] Y. Saad and H.A. van der Vorst. Iterative solution of linear systems in the 20-th century. *J. Comp. Appl. Math.*, 123:1–33, 2000.

[Tol99] S. Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In J.M. Abello and J.S. Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathemat-

ics and Theoretical Computer Science; Vol. 50, pages 161–179. American Mathematical Society Press, 1999.

[UD98] E. Uysal and T. Dayar. Iterative Methods Based on Splittings for Stochastic Automata Networks. *Eur. J. Op. Res.*, 110(1):166–186, 1998.

[Vit01] J.S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, 2001.