Technical Report

UCAM-CL-TR-650
ISSN 1476-2986

Number 650

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Parallel iterative solution method for large sparse linear equation systems

Rashid Mehmood, Jon Crowcroft

October 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Parallel iterative solution method for large sparse linear equation systems

Rashid Mehmood and Jon Crowcroft
University of Cambridge Computer Laboratory, Cambridge, UK.
Email: {rashid.mehmood, jon.crowcroft}@cl.cam.ac.uk

Abstract

Solving sparse systems of linear equations is at the heart of scientific computing. Large sparse systems often arise in science and engineering problems. One such problem we consider in this paper is the steady-state analysis of Continuous Time Markov Chains (CTMCs). CTMCs are a widely used formalism for the performance analysis of computer and communication systems. A large variety of useful performance measures can be derived from a CTMC via the computation of its steady-state probabilities. A CTMC may be represented by a set of states and a transition rate matrix containing state transition rates as coefficients, and can be analysed using probabilistic model checking. However, CTMC models for realistic systems are very large. We address this largeness problem in this paper, by considering parallelisation of symbolic methods. In particular, we consider Multi-Terminal Binary Decision Diagrams (MTBDDs) to store CTMCs, and, using Jacobi iterative method, present a parallel method for the CTMC steady-state solution. Employing a 24-node processor bank, we report results of the sparse systems with over a billion equations and eighteen billion nonzeros.

1 Motivation

Solving systems of linear equations is at the heart of scientific computing. Many problems in science and engineering give rise to linear equation systems, such as, forecasting, estimation, approximating non-linear problems in numerical analysis and integer factorisation: another example is the steady-state analysis of *Continuous Time Markov Chains* (*CTMCs*), a problem which we will focus on in this document.

Discrete-state models are widely employed for modelling and analysis of communication networks and computer systems. It is often convenient to model such systems as continuous time Markov chains, provided probability distributions are assumed to be exponential. A CTMC may be represented by a set of states and a transition rate matrix containing state transition rates as coefficients, and can be analysed using probabilistic model checking. Such an analysis proceeds by specifying desired performance properties as some temporal logic formulae, and by automatically verifying these properties using the appropriate model checking algorithms. A core component of these algorithms is the computation of the *steady-state probabilities* of the CTMC. This is reducible to the classical problem of solving a sparse system of linear equations, of the form $Ax = b$, of size equal to the number of states in the CTMC.

A limitation of the Markovian modelling approach is that the CTMC models tend to grow extremely large due to the state space explosion problem. This is caused by the fact that a system is usually composed of a number of concurrent sub-systems, and that the size of the state space of the overall system is generally exponential in the number of sub-systems. Hence, realistic systems can give rise to much larger state spaces, typically over 10^6. As a consequence, much research is focused on the development of techniques, that is, methods and data structures, which minimise the computational (space and time) requirements for analysing large and complex systems.

A standard approach for steady-state solution of CTMCs is to use *explicit methods* – the methods which store the state space and associated data structures using sparse storage techniques inherited from the linear algebra community. Standard numerical algorithms can thus be used for CTMC analysis. These explicit approaches typically provide faster solutions due to the fast, array-based data structures used. However, these can only solve models which can be accommodated by the RAM available in contemporary workstations. The so-called (explicit) *out-of-core* approaches [16, 39] have used disk memory to overcome the RAM limitations of a single workstation, and have made significant progress in extending the size of the solvable models on a single workstation. A survey of the out-of-core solutions can be found in [45].

Another approach for CTMC analysis comprises *implicit methods*. The so-called implicit methods can be traced back to *Binary Decision Diagrams* (*BDDs*) [6] and the *Kronecker* approach [56]. These rely on ex-

ploiting the regularity and structure in models, and hence provide an implicit, usually compact, representation for large models. Among these implicit techniques, the methods which are based on binary decision diagrams and extensions thereof are usually known as *symbolic methods*. Further details on Kronecker-based approaches and symbolic methods can be found in the surveys, [9] and [53], respectively. A limitation of the pure implicit approach is that it requires explicit storage of the solution vector(s). Consequently, the implicit methods have been combined with out-of-core techniques to address the vector storage limitations [40]. A detailed discussion and analysis of both the implicit and explicit out-of-core approaches can be found in [49].

Shared memory multiprocessors, distributed memory computers, workstation clusters and Grids provide a natural way of dealing with the memory and computing power problems. The task can be effectively partitioned and distributed to a number of parallel processing elements with shared or distributed memories. Much work is available on parallel numerical iterative solution of general systems of linear equations, see [4, 19, 58], for instance. Parallel solutions for Markov chains have also been considered: for explicit methods which have only used the primary memories of parallel computers, see e.g. [44, 1, 51, 10]; and, for a combination of explicit parallel and out-of-core solutions; see [37,36,5]. Parallelisation techniques have also been applied to the implicit methods. These include the Kronecker-based parallel approaches [8,21,35]; and the parallel approaches [43, 64], which are based on a modified form [49] of *Multi-Terminal Binary Decision Diagrams* (*MTBDDs*). MTBDDs [14, 3] are a simple extension of binary decision diagrams; these will be discussed in a later section of this paper.

In this paper, we consider parallelisation of the symbolic methods for the steady-state solution of CTMCs. In particular, for our parallel solution, we use the modified form of MTBDDs which was introduced in [49,47]. We chose this modified MTBDD because it provides an extremely compact representation for CTMCs while delivering solution speeds almost as fast as the sparse methods. Secondly, because (although it is symbolic) it exhibits a high degree of parallelism, it is highly configurable, and allows effective decomposition and manipulation of the symbolic storage for CTMC matrices. Third, because the time, memory, and decomposition properties for these MTBDDs have already been studied for very large models, with over a billion states; see [49].

The earlier work ([43,64]) on parallelising MTBDDs have focused on keeping the whole matrix (as a single MTBDD) on each computational node. We address the limitations of the earlier work by presenting a parallel solution method which is able to effectively partition, distribute, and manipulate the MTBDD-based symbolic storage. Our method, therefore, is scalable to address larger models.

We present a parallel implementation of the MTBDD-based steady-state solution of CTMCs using the Jacobi iterative method, and report solutions of models with over 1.2 billion states and 16 billion transitions (off-diagonal nonzeros in the matrix) on a processor bank. The processor bank which simply is a collection of loosely coupled machines, consists of 24 dual-processor nodes. Using three widely used CTMC benchmark models, we give a fairly detailed analysis of the implementation of our parallel algorithm employing up to 48 processors. Note that the experiments are performed without an exclusive access to the processor bank.

The rest of the paper is organised as follows. In Section 2, we give the background material which is related to this paper. In Section 3, we present and discuss a serial block Jacobi algorithm. In Section 4, in the context of our method we discuss some of the main issues in parallel computing, and describe our parallel algorithm and its implementation. The experimental results from the implementation, and its analysis is given in Section 5. In Section 6, the contribution of our work is discussed in relation to the other work on parallel CTMC solutions in the literature. Section 6 also gives a classification of the parallel solution approaches. Section 7 concludes and summarises future work.

2 Background Material

This section gives the background material. In Sections 2.1 to 2.4, and Section 2.6, we briefly discuss iterative solution methods for linear equation systems. In Section 2.5, we explain how the problem of computing steady-state probabilities for CTMCs is related to the solution of linear equation systems. Section 2.7 reviews the relevant sparse storage schemes. We have used MTBDDs to store CTMCs; Section 2.8 gives a short description of the data structure. Finally, in Section 2.9, we briefly introduce the case studies which we have used in this paper to benchmark our solution method. Here, using these case studies, we also give a comparison of the storage requirements for the main storage schemes.

2.1 Solving Systems of Linear Equations

Large sparse systems of linear equations of the form $Ax = b$ often arise in science and engineering problems. An example is the mathematical modelling of physical systems, such as climate modelling, over discretized domains. The numerical solution methods for linear systems of equations, $Ax = b$, are broadly classified into two categories: *direct methods*, such as *Gaussian elim-*

ination, *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 [4]. 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 [4,2,24,25,58,38]. In [59], Saad and Vorst present a survey of the iterative methods; [61] describes iterative methods in the context of solving Markov chains. A fine discussion of the parallelisation issues for iterative methods can be found in [58, 4].

2.2 Jacobi and JOR Methods

Jacobi method belongs to the category of so-called *stationary* iterative methods. These methods 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.

To solve a system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$, and $x, b \in \mathbb{R}^n$, the Jacobi method performs the following computations in its k-th iteration:

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

for all i, $0 \leq i < n$. In the equation, a_{ij} denotes the element in row i and column j of matrix A and, $x_i^{(k)}$ and $x_i^{(k-1)}$ indicate the i-th element of the iteration vector for the iterations numbered k and $k-1$, respectively. The Jacobi equation given above can also be written in matrix notation as:

$$x^{(k)} = D^{-1}(L+U)\; x^{(k-1)} \; + \; D^{-1}b, \tag{2}$$

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).

The Jacobi method does not converge for all linear equation systems. In such cases, Jacobi may be made to converge by introducing an *under-relaxation* parameter in the standard Jacobi. Furthermore, it may also be possible to accelerate the convergence of the standard Jacobi method by using an *over-relaxation* parameter. The resulting method is known as *Jacobi overrelaxation* (JOR) method. A JOR iteration is given by

$$x_i^{(k)} = \alpha\hat{x}_i^{(k)} + (1-\alpha)x_i^{(k-1)}, \tag{3}$$

for $0 \leq i < n$, where $\hat{x}$ denotes a Jacobi iteration as given by Equation (1), and $\alpha \in (0, 2)$ is the relaxation parameter. The method is under-relaxed for $0 < \alpha < 1$, and is over-relaxed for $\alpha > 1$; the choice $\alpha = 1$ reduces JOR to Jacobi.

Note in Equations (1) and (3), that the order in which the equations are updated is irrelevant, since the Jacobi and the JOR methods treat them independently. It can also be seen in the Jacobi and the JOR equations that the new approximation of the iteration vector ($x^{(k)}$) is calculated using only the old approximation of the vector ($x^{(k-1)}$). These methods, therefore, possess high degree of natural parallelism. However, Jacobi and JOR methods exhibit relatively slow convergence.

2.3 Gauss-Seidel and SOR

The Gauss-Seidel method typically converges faster than the Jacobi method by using the most recently available approximations of the elements of the iteration vector. The other advantage of the Gauss-Seidel algorithm is that it can be implemented using only one iteration vector, which is important for large linear equation systems where storage of a single iteration vector alone may require 10GB or more. However, a consequence of using the most recently available solution approximation is that the method is inherently sequential – it does not possess natural parallelism (for further discussion, see Section 4, Note 4.1). The Gauss-Seidel method has been used for parallel solutions of Markov chains, see [43, 64].

The successive over-relaxation (SOR) method extends the Gauss-Seidel method using a relaxation factor $\omega \in (0, 2)$, analogous to the JOR method discussed above. 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.

2.4 Krylov Subspace Methods

The *Krylov subspace methods* belong to the category of non-stationary iterative methods. These methods 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. For example, the *conjugate gradient squared* (CGS) method [60] performs 2 MVPs, 6 vector updates and two vector inner products during each iteration, and requires 7 iteration vectors.

The CGS method has been used for parallel solution of Markov chains, see [37, 5].

2.5 CTMCs and the Steady-State Solution

A CTMC is a continuous time, discrete-state stochastic process. More precisely, a CTMC is a *stochastic process* $\{X(t), t \geq 0\}$ which satisfies the *Markov property*:

$$\begin{aligned} P[X(t_k) = x_k | X(t_{k-1}) = x_{k-1}, \cdots, X(t_0) = x_0] \\ = P[X(t_k) = x_k | X(t_{k-1}) = x_{k-1}], \end{aligned} \quad (4)$$

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 [61].

Consider $Q \in \mathbb{R}^{n \times n}$ is the infinitesimal generator matrix of a continuous time Markov chain with n states, and $\pi(t) = [\pi_0(t), \pi_1(t), \ldots, \pi_{n-1}(t)]$ is 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 following differential equation:

$$\frac{d\pi(t)}{dt} = \pi(t)\, Q. \quad (5)$$

The initial probability distribution of the CTMC, $\pi(0)$, is also required to compute Equation (5). In this paper, we have focused on computing the steady-state behaviour of a CTMC. This is obtained by solving the following system of linear equations:

$$\pi Q = 0, \;\; \sum_{i=0}^{n-1} \pi_i = 1. \quad (6)$$

The vector $\pi = \lim_{t \rightarrow \infty} \pi(t)$ in Equation (6) is the *steady-state probability* vector. A sufficient condition for the unique solution of the Equation (6) is that the CTMC is finite and irreducible. A CTMC is *irreducible* if every state can be reached from every other state. In this paper, we consider solving only irreducible CTMCs; for details on the solution in the general case, see [61], for example. The Equation (6) 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 (see Section 2.1).

2.6 Test of Convergence for Iterative Methods

The *residual vector* of a system of linear equations, $Ax = b$, is defined by $\xi = b - Ax$. For an iterative method, the initial value for the residual vector, $\xi^{(0)}$, can be computed by $\xi^{(0)} \leftarrow b - Ax^{(0)}$, using some initial approximation of the solution vector, $x^{(0)}$. Through successive approximations, the goal is to obtain $\xi = 0$, which gives the desired solution x for the linear equation 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, some computations are performed in each iteration to test for convergence. A frequent choice for the convergence test is to compare, in the k-th iteration, the *Euclidean norm* of the residual vector, $\xi^{(k)}$, against some predetermined threshold, usually $\varepsilon \times \|\xi^{(0)}\|_2$ for $0 < \varepsilon \ll 1$. The Euclidean norm (also known as the l_2-*norm*) of the residual vector in the k-th iteration, $\xi^{(k)}$, is given by:

$$\|\xi^{(k)}\|_2 = \sqrt{\xi^{(k)^T} \xi^{(k)}}. \quad (7)$$

For further details on convergence tests, see e.g. [4, 58]. In the context of the steady-state solution of a CTMC, a widely used convergence criterion is the so-called *relative error* criterion (l_∞-*norm*):

$$\max_i \{ \mid x_i^{(k)} - x_i^{(k-1)} \mid \div \mid x_i^{(k)} \mid \} \; < \; \varepsilon \; \ll 1. \quad (8)$$

2.7 Explicit Storage Methods for Sparse Matrices

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. A number of sparse storage schemes exist which exploit various matrix properties, e.g., the *sparsity pattern* of a matrix. We briefly survey the notable sparse schemes in this section, with no intention of being exhaustive; for more schemes see, for instance, [4, 38]. A relatively detailed version of the review of the sparse storage schemes given here can also be found in [49].

2.7.1 The Coordinate and CSR Formats

The *coordinate format* [57, 32] is the simplest of sparse schemes. It makes no assumption about the matrix. The scheme uses three arrays to store an $n \times n$ sparse matrix. The first array `Val` stores the nonzero entries of the matrix in an arbitrary order. The nonzero entries include "a" off-diagonal matrix entries, and n entries in the diagonal. Therefore, the first array is of size $a+n$ doubles. The other two arrays, `Col` and `Row`, both of size $a+n$ ints, store the column and row indices for these nonzero 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.

The *compressed sparse row* (CSR) [57] format stores the $a+n$ nonzero matrix entries in the row by row order, in the array `Val`, and keeps the column indices of these entries in the array `Col`; the elements within a row are stored in an arbitrary 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.

2.7.2 Modified Sparse Row

Since many iterative algorithms treat the principal diagonal entries of a matrix differently, it is usually efficient to store the diagonal separately in an array of n doubles. Storage of column indices of diagonal entries in this case is not required, offering a saving of $4n$ bytes over the CSR format. The resulting storage scheme is known as the *modified sparse row* (MSR) format [57] (a modification of CSR). The MSR scheme essentially is the same as the CSR format except that the diagonal elements are stored separately. It requires $12(a+n)$ bytes to store the whole sparse matrix. For iterative methods, some computational advantage may be gained by storing the diagonal entries as $1/a_{ii}$ instead of a_{ii}, i.e. by replacing n division operations with n multiplications.

2.7.3 Avoiding the Diagonal Storage

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. This is accomplished as follows. 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 Jacobi.

2.7.4 Indexed MSR

The indexed MSR scheme exploits properties in CTMC matrices to obtain further space optimisations for CTMC storage. This is explained as follows. 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 among the off-diagonal entries of a matrix, with $MaxD \leq 2^{16}$; then *MaxD* distinct values can be stored as an array, `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`$[n]$.

The indexed variation of the MSR scheme (*indexed MSR*) 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* distinct 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. In addition, the indexed MSR scheme requires an array to store the actual (distinct) matrix values, `double Val`$[MaxD]$. The total memory requirement for this scheme (to store an off-diagonal matrix) 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. Note also that the indexed MSR scheme can be used to store matrix R, rather than Q, and therefore the diagonal storage during the iterative computation phase can be avoided. The indexed MSR scheme has been used in the litera-

ture with some variations, see [17, 37, 5].

2.7.5 Compact MSR

We note that the indexed MSR format stores the column index of a nonzero entry in a matrix as an int. An int usually uses 32 bits, which can store a column index as large as 2^{32}. The size of the matrices which can be stored within the RAM of a modern workstation are usually much smaller than 2^{32}. Therefore, the largest column index for a matrix requires fewer than 32 bits, leaving some spare bits. Even more spare bits can be made available for parallel solutions because it is a common practice (or, at least it is possible) to use per process local numbering for a column index. The *compact MSR* format [39,49] exploits these 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. The storage and retrieval of these indices into, and from, an int is carried out efficiently using bit operations. The scheme uses three arrays: the array `Col_i`$[a]$ of length $4a$ bytes which stores the column positions of matrix entries as well as the indices to these entries, the n-byte sized array `row_entries`$[n]$ to store the number of entries in each row, and the $2n$-byte sized array `Diag_i`$[n]$ of short indices to the original values in the diagonal. The total memory requirements for the compact MSR format is thus $4a + 3n$ bytes, around 30% more compact than the indexed MSR format.

2.8 Multi-Terminal Binary Decision Diagrams

We know from Section 1 that implicit methods are another well-known approach to the CTMC storage. These do not require data structures of size proportional to the number of states, and can be traced back to Binary Decision Diagrams (BDDs) [6] and the Kronecker approach [56]. Among these methods are multi-terminal binary decision diagrams (MTBDDs), Matrix Diagrams (MD) [11, 52] and the Kronecker methods (see e.g. [56, 20, 62, 7], and the survey [9]); *on-the-fly* method [18] can also be considered implicit because it does not require explicit storage of whole CTMC. We now will focus on MTBDDs.

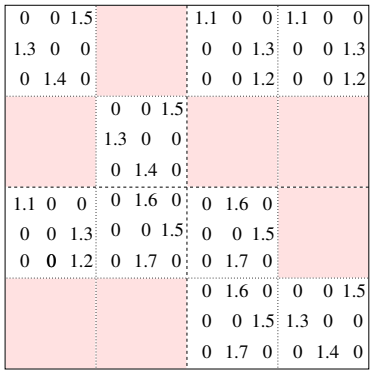
Multi-Terminal Binary Decision Diagrams (MTBDDs) [14, 3] are a simple extension of binary decision diagrams (BDDs). An MTBDD is a rooted, directed acyclic graph (DAG), which represents a function mapping Boolean variables to real numbers. MTBDDs can be used to encode real-valued vectors and matrices 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. The advantage of using MTBDDs (and other implicit 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.

Numerical solution of CTMCs can be performed purely using conventional MTBDDs; see for example [29, 31, 30]. This is done by representing both the matrix and the vector as MTBDDs and using an MTBDD-based matrix-vector multiplication algorithm (see [14, 3], 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 for an efficient implementation of the Gauss-Seidel iterative method[1]. An implementation of Gauss-Seidel is desirable because it typically converges faster than Jacobi and it requires only one iteration vector instead of two.

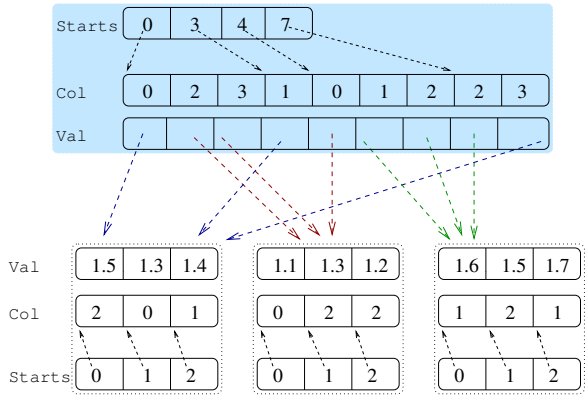
The limitations of the purely MTBDD-based approach mentioned above were addressed by *offset-labelled* MTBDDs [42, 55]. An explicit, array-based storage for the solution vector was combined with an MTBDD-based storage of the matrix, by adding offsets to the MTBDD nodes. This removed the vector irregularity problem. Nodes near the bottom of MTBDD were replaced by array-based (explicit) storage, which yielded significant improvements for the numerical solution speed. Finally, the offset-labelled MTBDDs allowed the use of the pseudo Gauss-Seidel iterative method, which typically converges faster than the Jacobi iterative method and requires storage of only one iteration vector.

Further improvements for (offset-labelled) MTBDDs have been introduced in [49, 47]. We have used this version of MTBDDs to store CTMCs for the parallel solution presented in this paper. The data structure in fact comprises a *two-layered* storage, made up entirely of sparse matrix storage schemes. It is, however, considered a symbolic data structure because it is constructed directly from the MTBDD representation and is reliant on the exploitation of regularity that this provides. This version of the MTBDDs is a significant improvement over its predecessors. First, it has relatively better time and memory characteristics. Second, the execution speeds for the (in-core) solutions based on this version are equal for different types of models (at least for the case studies considered). Conversely, the solution times for other MTBDD versions are dependent on the amount of structure in a model, typically resulting in much worse performance for the models which have less structure. Third, the data structure exhibits a higher degree of parallelism compared to

[1] In fact, an MTBDD version of the Gauss-Seidel method, using matrix-vector multiplication, has been presented in the literature [31]. However, this relies on computing and representing matrix inverses using MTBDDs. This will be inefficient in general, because converting a matrix to its inverse will usually result in a loss of structure and possibly fill-in.



(a) A CTMC matrix



(b) Modified MTBDD representation

Figure 1: A CTMC matrix and its representation as a modified MTBDD

its predecessors. It is highly configurable, and is entirely based on a fast, array-based storage, which allows effective decomposition, distribution, and manipulation of the data structure. Finally, the Gauss-Seidel method can be efficiently implemented using the modified MTBDDs, which is not true for its predecessors; see [47,49], for serial Gauss-Seidel, and [43,64], for parallel Gauss-Seidel implementations.

Figure 1 depicts a 12×12 CTMC matrix and its representation as a (modified) MTBDD. Note the two-layered storage. MTBDDs store the diagonal elements of a CTMC matrix separately as an array, in order to preserve structure in the symbolic representation of the CTMC. Hence, the diagonal entries of the matrix in Figure 1(a) are all zero. The matrix is divided into 4^2 blocks, where some of the blocks are zero (shown as shaded in pink). Each block in the matrix is of size 3×3. The blocks in the matrix are shown of equal size. However, usually, an MTBDD yields matrix blocks of unequal and varying sizes.

The information for the nonzero blocks in the matrix is stored in the MSR format, using the three arrays `Starts`, `Col`, and `Val` (see the top part of Figure 1(b)). The array `Col` stores the column indices of the matrix blocks in a row-wise order, and the i-th element of the array `Starts` contains the index in `Col` of the beginning of the i-th row. The array `Val` keeps track of the actual block storage in the bottom layer of the data structure. Each distinct matrix block is stored only once, using the MSR format; see bottom of Figure 1(b). The modified MTBDDs actually use the compact MSR format for both the top and the bottom layers of the storage. For simplicity, we used the (standard) MSR scheme in the figure.

The modified MTBDD can be configured to tailor the time and memory properties of the data structure according to the needs of the solution methods. This is explained as follows. An MTBDD is a rooted, directed acyclic graph, comprising two types of nodes: *terminal*, and *non-terminal*. Terminal nodes store the actual matrix entries, while non-terminal nodes store integers, called offsets. These offsets are used to compute the indices for the matrix entries stored in the terminal nodes. The nodes which are used to compute the row indices are called *row nodes*, and those used to compute the column indices are called *column nodes*. A *level* of an MTBDD is defined as an adjacent pair of *rank* of nodes, one for row nodes and the other for column nodes. Each rank of nodes in MTBDD corresponds to a distinct boolean variable. The total number of levels is denoted by l_{total}. Descending each level of an MTBDD splits the matrix into 4 submatrices. Therefore, descending l_{b} levels, for some $l_{\text{b}} < l_{\text{total}}$, gives a decomposition of a matrix into P^2 blocks, where $P = 2^{l_{\text{b}}}$. The decomposition of the matrix shown in Figure 1(a) into 16 blocks is obtained by descending 2 levels in the MTBDD, i.e. by using $l_{\text{b}} = 2$. The number of block levels l_{b} can be computed using $l_{\text{b}} = k \times l_{\text{total}}$, for some k, with $0 \leq k < 1$. A larger value for k (with some threshold value below 1) typically reduces the memory requirements to store a CTMC matrix, however, it yields larger number of blocks for the matrix. A larger number of blocks usually can worsen performance for out-of-core and parallel solutions.

In [49], the author used $l_{\text{b}} = 0.6 \times l_{\text{total}}$ for the in-core solutions, and $l_{\text{b}} = 0.4 \times l_{\text{total}}$ for the out-of-core solution. A detailed discussion of the affects of the parameter l_{b} on the properties of the data structure, a comprehensive description of the modified MTBDDs, and the analyses of its in-core and out-of-core implementations can be found in [49].

2.9 Case Studies

We have used three widely used CTMC case studies to benchmark our parallel algorithm. The case studies have been generated using the tool PRISM [41]. First among these is the flexible manufacturing system

Table 1: Comparison of Storage Methods

k	States (n)	Off-diagonal nonzeros (a)	a/n	Memory for Matrix (MB)				Vector (MB)
				MSR format	*Indexed MSR*	*Compact MSR*	*MTBDDs*	
FMS models								
6	537,768	4,205,670	7.82	50	24	17	4	4
7	1,639,440	13,552,968	8.27	161	79	53	12	12
8	4,459,455	38,533,968	8.64	457	225	151	29	34
9	11,058,190	99,075,405	8.96	1,176	577	388	67	84
10	25,397,658	234,523,289	9.23	2,780	1,366	918	137	194
11	54,682,992	518,030,370	9.47	6,136	3,016	2,028	273	417
12	111,414,940	1,078,917,632	9.68	12,772	6,279	4,220	507	850
13	216,427,680	2,136,215,172	9.87	25,272	12,429	8,354	921	1,651
14	403,259,040	4,980,958,020	12.35	58,540	28,882	19,382	1,579	3,077
15	724,284,864	9,134,355,680	12.61	107,297	52,952	35,531	2,676	5,526
Kanban models								
4	454,475	3,979,850	8.76	47	23	16	1	3.5
5	2,546,432	24,460,016	9.60	289	142	95	2	19
6	11,261,376	115,708,992	10.27	1,367	674	452	6	86
7	41,644,800	450,455,040	10.82	5,313	2,613	1,757	16	318
8	133,865,325	1,507,898,700	11.26	17,767	8,783	5,878	46	1,021
9	384,392,800	4,474,555,800	11.64	52,673	25,881	17,435	99	2,933
10	1,005,927,208	12,032,229,352	11.96	141,535	69,858	46,854	199	7,675
Polling System								
15	737,280	6,144,000	8.3	73	35	24	1	6
16	1,572,864	13,893,632	8.8	165	81	54	3	12
17	3,342,336	31,195,136	9.3	370	181	122	6	26
18	7,077,888	69,599,232	9.8	823	404	271	13	54
19	14,942,208	154,402,816	10.3	1,824	895	601	13	114
20	31,457,280	340,787,200	10.8	4,020	1,974	1,326	30	240
21	66,060,288	748,683,264	11.3	8,820	4,334	2,910	66	504
22	138,412,032	1,637,875,712	11.8	19,272	9,478	6,362	66	1,056
23	289,406,976	3,569,352,704	12.3	41,952	20,645	13,855	144	1,081
24	603,979,776	7,751,073,792	12.8	91,008	44,813	30,067	144	1,136
25	1,258,291,200	16,777,216,000	13.3	196,800	96,960	65,040	317	1,190

(FMS) of Ciardo and Tilgner [13], who used this model to benchmark their decomposition approach for the solution of large stochastic reward nets (SRNs), a class of Markovian stochastic Petri nets [54]. 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 type of 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. Second CTMC model is the Kanban manufacturing system [12], again due to Ciardo and Tilgner. The authors used the Kanban model 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. Finally, third CTMC model is the cyclic server polling system of Ibe and Trivedi [33]. 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. We will abbreviate the names of these case studies to "FMS", "Kanban" and "Polling" respectively.

Table 1 gives statistics for the three CTMC models, and compares storage requirements for MSR, indexed MSR, compact MSR and (modified) MTBDDs. The first column in the table gives the model parameter k; 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. The largest model reported in the table is Polling ($k = 25$) with over 12.58 billion states and 16.77 billion transitions.

Columns $5 - 8$ in Table 1 give the (RAM) storage requirements for the CTMC matrices (excluding the storage for the diagonal) in MB for the four data structures. The indexed MSR format does not require RAM to store the diagonal for the iteration phase (see Section 2.7.3). The (standard) MSR format stores the diagonal as an array of $8n$ bytes. The compact MSR scheme and the (modified) MTBDDs store the diagonal entries as short int (2 Bytes), and therefore require an array of $2n$ bytes for the diagonal storage. The last column lists the memory required to store a single iteration vector of doubles (8 bytes) for the solution phase.

Note in Table 1 that the storage for the three explicit schemes is dominated by the memory required to store the matrix, while the memory required to store vector dominates the storage for the implicit scheme (MTBDD). Note also that the memory requirements for the three explicit methods are independent

of the case studies used, while for MTBDDs (implicit method), memory required to store CTMCs is heavily influenced by the case studies used. Finally, we observe that the memory required to store some of the Polling CTMC matrices is the same (e.g., $k = 23,\ 24$). This is possible in case of MTBDDs because these exploit structure in models which can possibly lead to similar amount of storage for different sizes of CTMCs.

The storage listed for the MTBDDs in Table 1 needs further clarification. We have mentioned earlier in Section 2.8 that the memory requirements for the MTBDDs can be configured using the parameter l_{b}. The memories for the MTBDD listed in the table are given for $l_{\mathrm{b}} = 0.4 \times l_{\mathrm{total}}$. For parallel solutions, we believe that this heuristic gives an adequate compromise between the amount of memory required for matrix storage and the number of blocks in the matrix. However, this issue needs further analysis and will be considered in our future work.

3 A Block Jacobi Algorithm

Iterative algorithms for the solution of linear equation system $Ax = b$ perform matrix computations in row-wise or column-wise fashion. Block-based formulations of the iterative methods which perform matrix computations on block by block basis usually turn out to be more efficient.

We have described Jacobi iterative method in Section 2.2. In this section, we present a block Jacobi algorithm for the solution of the linear equation system $Ax = b$. Using this algorithm, we will be able to compute the steady-state probabilities of a CTMC, with $A = Q^T$, $x = \pi^T$, and $b = 0$. In the following, we briefly explain the *block iterative methods*, and subsequently go on to describe our block Jacobi algorithm. For further details on block iterative methods, see e.g. [61, 15].

A block iterative method partitions a system of linear equations into a certain number of blocks or subsystems. We consider a decomposition 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 $n = \sum_{i=0}^{P-1} n_i$. We additionally define $n_{\max} = \max\{n_i \mid 0 \leq i < P\}$, the size of the largest CTMC partition (or equally, the largest block of the vector x). Using this decomposition of the state space, 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 $n_i \times n_j$. Therefore, for $P = 4$, the system of equations $Ax = b$ can be partitioned as follows.

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{9}$$

Algorithm 1 *A (Serial) Block Jacobi Algorithm*

ser_block_Jac($A,\ b,\ x,\ P,\ n_{[\,]},\ \varepsilon$) {

1. var $\tilde{x},\ Y,\ k \leftarrow 0,$ error $\leftarrow 1.0,\ i,\ j$
2. while(error $> \varepsilon$)
3. $k \leftarrow k + 1$
4. for($0 \leq i < P$)
5. $Y \leftarrow B_i$
6. for($0 \leq j < P; j \neq i$)
7. $Y \leftarrow Y - A_{ij} X_j^{(k-1)}$
8. vec_update_Jac($X_i^{(k)}, A_{ii}, X_i^{(k-1)}, Y, n_{[i]}$)
9. compute error
10. $X_i^{(k-1)} \leftarrow X_i^{(k)}$

}

Using a partitioning of the system $Ax = b$, as described above, a block iterative method solves P sub-systems of linear equations, of sizes $n_0, \ldots, n_{P-1}$, within a *global* iterative structure. If the Jacobi iterative method is employed as the global structure, it is called the *block Jacobi* method. From Equations (1) and (9), the block Jacobi method for the solution of the system $Ax = b$ is given by:

$$A_{ii}\ X_i^{(k)} = B_i - \sum_{j \neq i} A_{ij}\ X_j^{(k-1)}, \tag{10}$$

for all i, $0 \leq i < P$, 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.

Consequently, in the i-th of the total P phases of the k-th iteration of the block Jacobi iterative method, we solve Equation (10) for $X_i^{(k)}$. These sub-systems can be solved using either direct or iterative methods. It is not necessary even to use the same method to solve each sub-system. If iterative methods are used to solve these sub-systems then we may have several *inner* iterative methods, within a global or *outer* iterative method. Moreover, each of the P sub-systems of equations can receive either a fixed or varying number of *inner* iterations. The block iterative methods which employ an inner iterative method typically require fewer (outer) iterations, provided multiple (inner) iterations are applied to the sub-systems. However, a consequence of multiple inner iterations is that each outer iteration will require more work. Note that applying one Jacobi iteration on each sub-system in the global Jacobi iterative structure reduces the block Jacobi method to the standard Jacobi method, i.e., gives a block-based formulation of the (standard) Jacobi iterative method.

Algorithm 1 gives a block Jacobi algorithm for the solution of the system $Ax = b$. The algorithm, ser_block_Jac($\cdot$), accepts the following parameters as input: the references to matrix A and the vector b; the reference to the vector x, which contains an initial

approximation for the iteration vector; the number of partitions P; the vector $n_{[\,]}$; and, a precision value for the convergence test, ε. The vector $n_{[\,]}$ is of size P, and its i-th element contains the size of the i-th vector block. According to the notation introduced earlier for the block methods, $n_{[i]}$ equals n_i.

The local vectors and variables for the algorithm are declared, and initialised on line 1. The vectors x and $\tilde{x}$ are used for the two iteration vectors, $x^{(k-1)}$ and $x^{(k)}$, respectively. The notation used for the block methods applies to the algorithm: $X_i^{(k)}$, $X_i^{(k-1)}$ and B_i are the i-th blocks of vectors $x^{(k)}$, $x^{(k-1)}$ and b, respectively.

Each iteration of the algorithm consists of P phases of computations (see the outer `for` loop given by lines $4-8$). The i-th phase updates the elements from the i-th block (X_i) of the iteration vector, using the entries from the i-th row of blocks in A, i.e., A_{ij} for all j, $0 \leq j < P$. There are two main computations performed in each of the P phases: the computations given by lines $6-7$, where matrix-vector products (MVPs) are accumulated for all the matrix blocks of the i-th block row, except the diagonal block (i.e., $i \neq j$); and the computation given by line 8, which invokes the function vec_update_Jac$(\cdot)$ to update the i-th vector block.

The function vec_update_Jac$(\cdot)$ uses the Jacobi method to update the iteration vector blocks, and is defined by Algorithm 2. In the algorithm, $X_{i[p]}$ denotes 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}. Note that line 3 in Algorithm 2 updates p-th element of $X_i^{(k)}$ by accumulating the product of the vector $X_i^{(k-1)}$ and the p-th row of the block A_{ii}.

Algorithm 2 *A Jacobi (inner) Vector Update*

vec_update_Jac($X_i^{(k)}$, A_{ii}, $X_i^{(k-1)}$, Y, $n_{[i]}$){

1. `var` p, q
2. `for`($0 \leq p < n_{[i]}$)
3. $\quad X_i^{(k)}{}_{[p]} \leftarrow A_{ii[p,p]}{}^{-1}(Y_{[p]} - \sum_{q \neq p} A_{ii[p,q]} X_i^{(k-1)}{}_{[q]})$

}

3.1 Memory Requirements

Algorithm 1 requires storage for matrix A. To obtain an efficient implementation, the matrix must be stored in a format such that the matrix elements can be accessed in a block-row-wise fashion. Furthermore, the storage format should allow row-wise access to each block of the matrix. In addition to the matrix, the block Jacobi algorithm requires two arrays to store the iteration vectors, each of size n. Another array Y of size $n_{\max}$ is required to accumulate the MVPs on line 7 of the algorithm.

4 Parallelisation

We begin this section with a brief introduction to some of the issues in parallel computing in the context of iterative methods. Subsequently, we describe our parallel Jacobi algorithm and its implementation.

The design of parallel algorithms involves *partitioning* of the problem – the identification and the specification of the overall problem as a set of tasks that can be performed concurrently. A goal when partitioning a problem into several tasks is to achieve *loadbalancing*, such that no process waits for another while carrying out individual tasks. The communication among processes is usually expensive and hence it should be minimised, wherever possible, and overlapped with the computations. These goals may be at odds with one another. Hence, one should aim to achieve a good compromise between these conflicting demands.

Some computational problems naturally possess high degree of parallelism for concurrent scheduling, for example, the Jacobi iterative method. We know from Section 2.2, that in the Jacobi method, the order in which the linear equations are updated is irrelevant, since Jacobi treats them independently. The elements of the iteration vector, therefore, can be updated concurrently. Some problems, however, are inherently sequential. The Gauss-Seidel iterative method is a such example. It uses the most recently available approximation of the iteration vector (see Section 2.3). For sparse matrices, however, it is possible to parallelise the Gauss-Seidel method using *wavefront* or *multicoloring* techniques.

Multicolor ordering and wavefront techniques have been in use to extract and improve parallelism in iterative solution methods. For sparse matrices, these techniques can be used to reorder the iterative computations such that the computations of unrelated vector elements can be carried out in parallel. The *wavefront* technique partitions a linear equation system into wavefronts. The unknown elements within a wavefront can be computed asynchronously by assigning these to multiple processors. *Multicolor ordering* techniques can take this further if the aim is to maximise parallelism. The ordering refers to a technique of coloring the nodes of a graph associated with a matrix in such a way that no two adjacent nodes have the same color. The aim herein is to minimise the number of colors, and finding such minimum colorings is a combinatorial problem of exponential complexity. For iterative methods, however, simple heuristics can provide acceptable colorings. A Gauss-Seidel iteration, preconditioned with multicolor ordering, can proceed in phases equal to the number of colors used in the ordering, while computation of the unknown elements within a color can proceed in parallel. Note that the solution of a permuted system with parallel Gauss-Seidel does not always result in improved overall performance, because it is possible that the Jacobi method delivers better performance by exploiting the natural ordering of a particular sparse system. Furthermore, as a result of multicolor ordering, the rate of convergence for the permuted matrix is likely to decrease. For further details, see e.g. [4, 58].

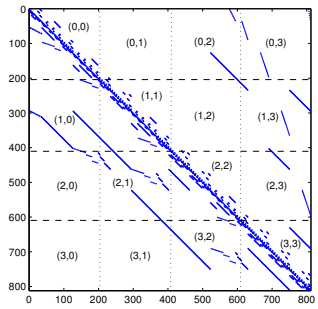
Note 4.1: Wavefronts, Multicoloring, and Gauss-Seidel

Parallelising the iterative methods involves decomposition of the matrix and the solution vector into blocks, such that the individual partitions can be computed concurrently. A number of schemes have been in use to partition a system. The *row-wise block-striped* partitioning scheme is one such example. The scheme decomposes a matrix into blocks of complete rows, and each process is allocated one such block. The vector is also decomposed into blocks of corresponding sizes.

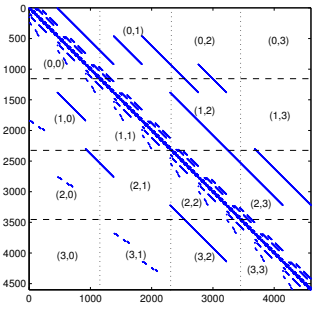
Figure 2 plots the off-diagonal nonzero entries for three CTMC matrices, one from each of the three case studies. The figure also depicts the row-wise block-striped partitioning for the three matrices. The matrices have been row-wise decomposed into four blocks (see dashed lines), for distribution to a total of four processes. Process p keeps all the (p, q)-th matrix blocks, for all q. Using this matrix partitioning, the iteration vector can also be divided into four blocks of corresponding sizes. Process p will be allocated, and made responsible to iteratively compute the p-th vector block. Note that, to compute the p-th vector block, process p will require access to all the q-th blocks such that $A_{pq} \neq \mathbf{0}$. Hence, we say that the p-th block for a process p is its *local* block, while all the q-th blocks, $q \neq p$, are the *remote* blocks for the process.

Matrix sparsity usually leads to poor loadbalancing. For example, balancing the computational load for parallel processes participating in the computations may not be easy, and/or the processes may have different communication needs. Sparsity in matrix also leads to high communication to computation ratio. For these reasons, parallelising sparse matrix computations compared to dense matrices is usually considered hard. However, it may be possible, sometimes, to exploit *sparsity pattern* of a matrix. For example, *banded* and *block-tridiagonal* matrices possess a fine degree of parallelism in their sparsity patterns. This is explained further in the following paragraph, using the CTMC matrices from the three case studies.

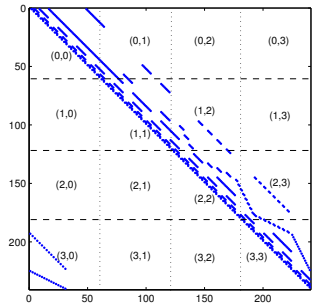
Consider again Figure 2, which also shows the sparsity pattern for three CTMC matrices. The sparsity pattern for all the matrices of a particular CTMC model is similar. Note in the figure that both FMS and Kanban matrices have irregular sparsity patterns, while the Polling matrix exhibits a regular pattern. Most of the nonzero entries in the Polling matrix are confined to a band on top of the principal diagonal, except those relatively few nonzero entries which are located within the lower left block of the matrix. If the matrix is partitioned among 4 processes, as shown in the figure, communication is only required in between two neighbouring processes. Furthermore, since most of the nonzero entries are located within the diagonal blocks, relatively few elements of the remote vector blocks are required. It has been shown in [48], that a parallel algorithm tailored in particular for the



(a) FMS ($k = 2$)



(b) Kanban ($k = 2$)



(c) Polling ($k = 5$)

Figure 2: Sparsity pattern for three CTMC matrices selected from the three case studies

sparsity pattern of the Polling matrices achieves significantly better performance than for arbitrary matrices. The parallel algorithm presented in this paper does not *explicitly* exploit sparsity patterns in matrices. However, we will see in Section 5 that the performance of the parallel algorithm for the Polling matrices is better than for the other two case studies.

4.1 The Parallel Algorithm

Algorithm 3 presents a high-level description of our parallel Jacobi iterative algorithm. It is taken with some modifications from our earlier work [48]. The algorithm employs the Jacobi iterative method for the

Algorithm 3 *A Parallel Jacobi for Process p*

par_block_Jac($\check{A}_p,\ D_p,\ B_p,\ X_p,\ T, N_p,\ \varepsilon$) {

1. var $\tilde{X}_p,\ Z,\ k \leftarrow 0,$ error $\leftarrow 1.0,\ q,\ h,\ i$
2. while(error $> \varepsilon$)
3. $\quad k \leftarrow k+1$
4. $\quad h = 0$
5. $\quad$ for($0 \leq q < T; q \neq p$)
6. $\quad\quad$ if($\check{A}_{pq} \neq \mathbf{0}$)
7. $\quad\quad\quad$ send(request$_{X_q}, q$)
8. $\quad\quad\quad h = h+1$
9. $\quad Z \leftarrow B_p - \check{A}_{pp} X_p^{(k-1)}$
10. $\quad$ while($h > 0$)
11. $\quad\quad$ if(probe(message))
12. $\quad\quad\quad$ if(message $=$ request$_{X_p}$)
13. $\quad\quad\quad\quad$ send(X_p, q)
14. $\quad\quad\quad$ else
15. $\quad\quad\quad\quad$ receive(X_q, q)
16. $\quad\quad\quad\quad Z \leftarrow Z - \check{A}_{pq} X_q^{(k-1)}$
17. $\quad\quad\quad\quad h = h-1$
18. $\quad$ serve(X_p, request$_{X_p}$)
19. $\quad$ for($0 \leq i < N_p$)
20. $\quad\quad {X_p^{(k)}}_{[i]} \leftarrow {D_{p[i]}}^{-1} Z_{[i]}$
21. $\quad$ compute error collectively

}

solution of the system of n linear equations, of the form $Ax = b$. The steady-state probabilities can be calculated using this algorithm with $A = Q^T$, $x = \pi^T$, and $b = 0$. The algorithm uses the single program multiple data (SPMD) paradigm – all the nodes execute the same binaries but operate on different sets of data. Note that the design of Algorithm 3 is influenced by the fact that MPI is thread-unsafe, otherwise concurrency in the algorithm can easily be improved.

We store the off-diagonal CTMC matrix using the (modified) MTBDDs [49,47] (see Section 2.8). The diagonal entries of the matrix are stored separately as an array. For convenience in this discussion, we additionally introduce: the matrix $\check{A}$, which contains the off-diagonal elements of A, with $\check{a}_{ii} = 0$, for all $0 \leq i < n$; and the diagonal vector d, with the entries $d_i = a_{ii}$.

Consider a total of T processes[2]. Algorithm 3 assumes that the off-diagonal matrix $\check{A}$ is row-wise block-striped partitioned into T contiguous blocks of complete rows, of sizes $N_0, \ldots, N_{T-1}$, such that $n = \sum_{p=0}^{T-1} N_p$. A such partitioning is depicted in Figure 2. Process p is allocated a matrix block of size $N_p \times n$, with all the rows numbering from $\sum_{q=0}^{p-1} N_q$ to $\sum_{q=0}^{p} N_q - 1$. Moreover, each row of blocks is further divided into T blocks such that the p-th process keeps all the blocks, $\check{A}_{pq},\ 0 \leq q < T$. We will use $\check{A}_p$ to denote the block row containing all the blocks $\check{A}_{pq}$.

Using the same partitioning as for the matrix, the iteration vector x, and the diagonal vector d, are divided into T blocks each, of the sizes, $N_0, \ldots N_{T-1}$. Process p keeps the p-th blocks, X_p and D_p, of the two vectors and is responsible for updating the iteration vector block X_p during each iteration. As mentioned earlier, we will refer to the block X_p as the *local* block for process p because it does not require communication while computing the MVP $\check{A}_{pp} X_p^{(k-1)}$. Conversely, all the blocks $X_q,\ q \neq p$, are the *remote* blocks for process p because computing $\check{A}_{pq} X_q^{(k-1)}$ requires blocks $X_q^{(k-1)}$ which are owned by the other processes.

The algorithm, par_block_Jac($\cdot$), accepts the following parameters as input: the reference to the p-th row of the off-diagonal matrix blocks, $\check{A}_p$; the reference to the p-th diagonal block, D_p; the reference to the block B_p which is zero in our case; the reference to the vector block X_p which contains an initial approximation for the iteration vector block; the total number of processes T; the constant N_p, which gives the number of entries in the vector block X_p; and a precision value for the convergence test, ε.

The vectors and variables local to the algorithm are declared, and initialised on line 1. The vectors X_p and $\tilde{X}_p$ are used in the algorithm for the two iteration vectors, $X_p^{(k-1)}$, $X_p^{(k)}$, respectively.

In each Jacobi iteration, given in Algorithm 3, the main task of process p is to accumulate the MVPs, $\check{A}_{pq} X_q$, for all q (see lines 9 and 16). These MVPs are accumulated using the vector Z. Having accomplished this, the local block X_p for process p can be computed using the vectors Z and D_p (lines $19-20$). Since process p does not have access to all the blocks X_q, except for $q = p$, it sends requests to all the processes numbered q for the blocks X_q corresponding to the nonzero matrix blocks $\check{A}_{pq}$ (lines $4-8$). The process computes the MVP for local block on line 9. Subsequently, the process iteratively probes for the messages from other processes (lines $10-17$). The block X_p is sent to process q if a request from the process is received (lines $12-13$)). If process q has sent the block X_q, the block is received and the MVP for this remote block is computed (lines $15-17$). Once all the MVPs have been accumulated, process p waits and serves for any remaining requests for X_p from other processes (line 18), and then goes on to update its local block X_p for the k-th iteration (lines $19-20$). Finally, on line 21, all the processes collectively perform the test for convergence – each process p computes error using the criteria given by Equation (8), for $0 \leq i < N_p$, and the maximum of all these errors is determined and communicated to all the processes.

The CTMC matrices for the Kanban model do not

[2] In this paper, we have used process, rather than processor or node, because it has a more general meaning.

Table 2: Solution Results for Parallel Execution on 24 Nodes

k	States (n)	a/n	Memory/Node (MB)	Time		Total Iterations
				Iteration (seconds)	Total (hr:min:sec)	
FMS Model						
6	537,768	7.8	3	.04	39	1080
7	1,639,440	8.3	4	.13	2:44	1258
8	4,459,455	8.6	10	.23	5:31	1438
9	11,058,190	8.9	22	.60	16:12	1619
10	25,397,658	9.2	47	1.30	39:06	1804
11	54,682,992	9.5	92	2.77	1:31:58	1992
12	111,414,940	9.7	170	6.07	3:40:57	2184
13	216 427 680	9.9	306	13.50	8:55:17	2379
14	403,259,040	10.03	538	25.20	18:02:45	2578
15	724,284,864	10.18	1137	48.47	37:26:35	2781
Kanban System						
4	454,475	8.8	2	.02	11	466
5	2,546,432	9.6	7	.16	1:46	663
6	11,261,376	10.3	17	.48	7:08	891
7	41,644,800	10.8	53	1.73	33:07	1148
8	133,865,325	11.3	266	5.27	2:02:06	1430
9	384,392,800	11.6	564	14.67	7:03:29	1732
10	1,005,927,208	11.97	1067	37.00	21:04:10	2050
Polling System						
15	737,280	8.3	1	.02	11	657
16	1,572,864	8.8	7	.03	24	709
17	3,342,336	9.3	14	.08	58	761
18	7,077,888	9.8	28	.14	1:57	814
19	14,942,208	10.3	55	.37	5:21	866
20	31,457,280	10.8	106	.77	11:49	920
21	66,060,288	11.3	232	1.60	25:57	973
22	138,412,032	11.8	328	2.60	44:31	1027
23	289,406,976	12.3	667	5.33	1:36:02	1081
24	603,979,776	12.8	811	11.60	3:39:38	1136
25	1,258,291,200	13.3	1196	23.97	7:54:25	1190

converge with the Jacobi iterative method. For Kanban matrices, hence, on line 20, we apply additional (JOR) computations given by Equation (3).

4.1.1 Implementation Issues

For each process p, Algorithm 3 requires two arrays of N_p doubles to store its share of the vectors for the iterations, $k-1$ and k. Another array of at most $n_{\max}$ (see Section 3) doubles is required to store the blocks X_q received from process q. Moreover, each process also requires storage of its share of the off-diagonal matrix $\check{A}$, and the diagonal vector D_p. The number of the distinct values in the diagonal of the CTMC matrices considered is relatively small. Therefore, storage of N_p short int indices to an array of the distinct values is required, instead of N_p doubles.

Note also that the modified MTBDD partitions a CTMC matrix into P^2 blocks for some P, as explained in Section 3. For the parallel algorithm, we make the additional restrictions that $T \leq P$, which implies that each process is allocated with at least one row of MTBDD blocks. In practice, however, the number P is dependent on the model and is considerably larger than the total number of processes T.

5 Experimental Results

In this section, we analyse performance for Algorithm 3 using its implementation on a processor bank. The processor bank is a collection of loosely coupled machines. It consists of 24 dual-processor nodes. Each node has 4GB of RAM. Each processor is an AMD Opteron(TM) Processor 250 running at 2400MHz with 1MB cache. The nodes in the processor bank are connected by a pair of Cisco 3750G-24T switches at 1Gbps. Each node has dual BCM5703X Gb NICs, of which only one is in use. The parallel Jacobi algorithm is implemented in C language using the MPICH implementation [27] of the message passing interface (MPI) standard [22]. The results reported in this section were collected without an exclusive access to the processor bank.

We analyse our parallel implementation with the help of the three benchmark CTMC case studies: FMS, Kanban and Polling systems. These have been introduced in Section 2.9. We use the PRISM tool [41] to generate these models. For our current purposes, we have modified version 1.3.1 of the PRISM tool. The modified tool generates the underlying matrix of a CTMC in the form of the modified MTBDD (see Section 2.8). It decomposes the MTBDD into partitions, and exports the partitions for parallel solutions.

The experimental results are given in Table 2. The

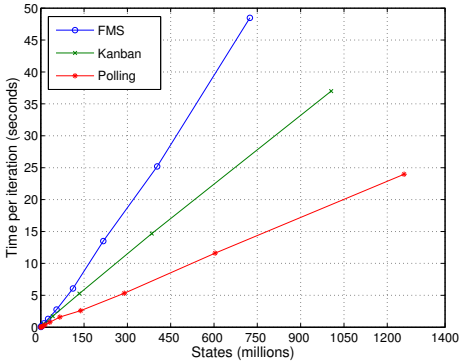
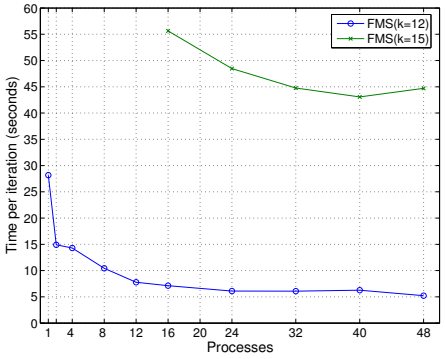


Figure 3: Comparison of the execution times per iteration for the three case studies (plotted against the number of states)

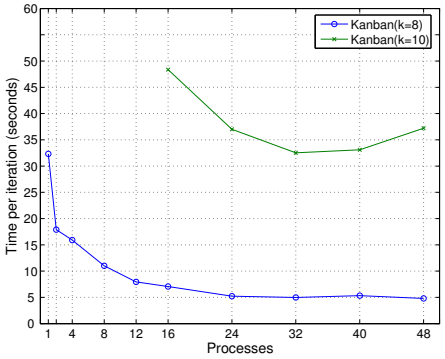
time and space statistics reported in this table are collected by executing the parallel program on 24 processors, where each processor pertains to a different node. For each CTMC, the first three columns in the table report the model statistics: column 1 gives the values for the model parameter k, column 2 lists the resulting number of reachable states (n) in the CTMC matrix, and column 3 lists the average number of nonzero entries per row (a/n). Column 4 reports the maximum of the memories used by the individual processes. The last three columns, $5-7$, report the execution time per iteration, the total execution time for the steady-state solution, and the number of iterations, respectively. All reported run times are wall clock times. For FMS and Polling CTMCs, the reported iterations are for the Jacobi iterative method, and for Kanban, the column gives the number of JOR iterations (Kanban matrices do not converge with Jacobi). The iterative methods were tested for the convergence criterion given by Equation (8) for $\varepsilon = 10^{-6}$.

In Table 2, the largest CTMC model for which we obtain the steady-state solution is the Polling system ($k = 25$). It consists of over 1258 million reachable states. The solution for this model used a maximum of 1196MB RAM per process. It took 1190 Jacobi iterations, and 7 hours, 54 minutes, and 25 seconds, to converge. Each iteration for the model took an average of 23.97 seconds. The largest Kanban model ($k = 10$) which was solved contains more than 1005 million states. The model required 2050 JOR iterations, and approximately 21 hours (less then a day) to converge, using no more than 1.1GB per process. The largest FMS model which consists of over 724 million states, took 2781 Jacobi iterations and less than 38 hours (1.6 days) to converge, using at most 1137MB of memory per process.

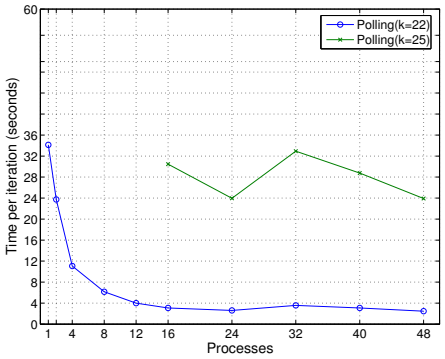
A comparison of the solution statistics for the three models in Table 2 reveals that the Polling CTMCs offer the fastest execution times per iteration, the FMS matrices exhibit the slowest times, and the times per



(a) FMS (times per iteration)



(b) Kanban (times per iteration)



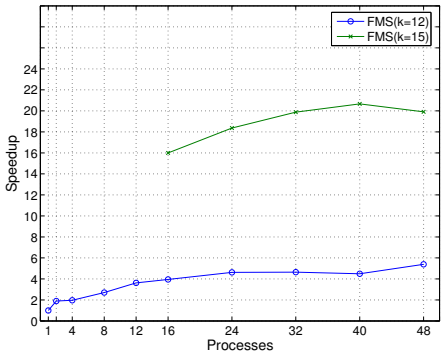
(c) Polling (times per iteration)

Figure 4: Time per iteration against the number of processes for six CTMCs, two from each case study

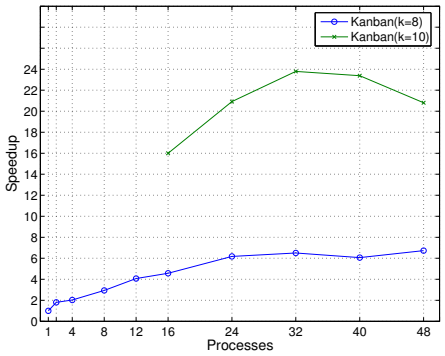
iteration for the Kanban CTMCs lie in between. Note also that the convergence rate for the Polling CTMCs is the highest (e.g. 1190 iterations) among the three types of models, while for FMS, it is the lowest (2781 iterations). To further explore the relative performance of our parallel solution for the three case studies, we plot the run times per iteration for the three example CTMCs against the number of states in Figure 3. In the figure, the magnitudes of the individual slopes for the three models expose their comparative speeds. A potential cause for the differences in the performance lies in that the three models possess different

amount of structure. The FMS system is the least structured of the three models. Since an MTBDD exploits model structure to produce a compact storage for CTMCs, models with less structure yield a larger MTBDD (see [49], Chapter 5, for a detailed discussion of the MTBDD). A larger MTBDD may require additional overhead for its parallelisation. However, a more important and stronger cause for this behaviour is the differences in the sparsity patterns of the three models; see Figure 2. Note in the figure that, among the three case studies, the FMS model exhibits the most irregular sparsity pattern. An irregular sparsity pattern can lead to load imbalance for computations as well as communications, and, as a consequence, can cause an overall worse performance.

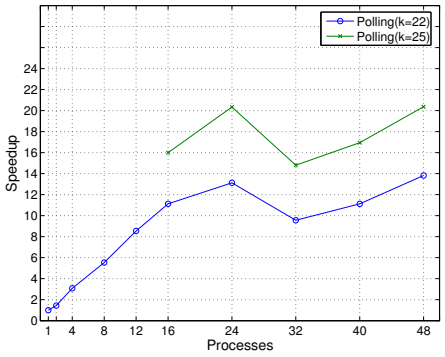
We now analyse performance for the parallel solution with respect to the number of processes. In Figure 4, for the three case studies, we plot the execution times per iteration against the number of processes. We select two models from each case study. One of these is the largest model solved for each case study; e.g., the Kanban ($K = 10$) model with 1005 million states. The execution times are collected for up to a maximum of 48 processes. For the larger models, however, run time are only collected for 16 or more processes due to their excessive RAM requirements. To explain the figure, we consider the FMS plots given in Figure 4(a), in particular, the plot for the (smaller) FMS model ($k = 12$), which contains over 111 million states. As expected, the increase in the number of processes (from 1 to 24) causes a decrease in the time per iteration (from 28.18 to 6.07 seconds). However, there is a somewhat continual drop in the value of the magnitude of the slope for the plot as it approaches the 24 process mark. Similarly, in Figure 4(a), the plot for the larger FMS model ($k = 15$, 724 million states) also shows a decrease in the execution times with an increase in the number of processes: increasing the processes from 16 to 24 brings the execution time from 55.63 seconds down to 48.47 seconds. However, for the larger model, the magnitude of the slope is greater (at this point of the plot), implying a steeper drop in the execution time.



(a) Speedups for the FMS model



(b) Speedups for the Kanban system



(c) Speedups for the Polling system

Figure 5: Speedup plotted against the number of processes for six CTMCs

Perhaps *speedup* is the most common measure in practice for evaluating performance of parallel algorithms. It captures the relative benefits of solving a problem in parallel. Speedup may be defined as the ratio of the time taken to solve a problem using a single process to the time required to solve the same problem using a collection of T concurrent processes. For a fair comparison, each parallel process must be given resources (CPU, RAM, I/O) identical to the resources given to the single process. Suppose, $\texttt{Time}_1$ and $\texttt{Time}_T$ are the times taken by the serial algorithm and the parallel algorithm on T processes, respectively. The speedup $\texttt{Speed}_T$ is given by $\texttt{Time}_1/\texttt{Time}_T$. The ideal speedup for T processes is T. Another measure to evaluate performance of parallel algorithms is *efficiency*, which is defined as the ratio of the speedup to the number of processes used, i.e., $\texttt{Speed}_T/T$. The ideal value for efficiency is 1, or 100%, although *superlinear* speedup can cause even higher values for efficiency.

Note 5.1: Speedup and Efficiency

To further investigate the parallel performance, in Figure 5, we plot speedups for six CTMCs against the number of up to 48 processes. We use the same six CTMCs which were used in Figure 4. Consider in Figure 5(b), the plot for the smaller Kanban model ($k = 8$, 133 million states). The plot shows that increasing the number of processes from 1 to 2 gives a speedup of 1.8, which equates to an efficiency of 0.90 or 90%. The plot, however, further reveals that a continual increase in

the number of processess will not be able to maintain the speedup, and will actually result in a somewhat steady decrease in the efficiency of the parallel solution. For example, the efficiency for 24 processes is 25.75%, which equates to $\texttt{Speed}_{24} = 6.18$. Similarly, using the plots in Figure 5, the speedup on 24 processes for the smaller FMS and Polling CTMCs, FMS($k = 12$) and Polling($k = 22$), can be calculated as 4.62 and 13.12, which equates to the efficiencies of 19.25% and 54.70%, respectively.

We now examine the plots for the larger models in Figure 5(b). Consider the plot for the Kanban model ($k = 10$, 1 billion states). Note in the plot that the (minimum) speedup[3] for 16 processes is 16. Using 48.37 (see Figure 4(b) for the value of the time per iteration) as the base time reference in the plot, the speedup on 24 processes can be calculated as $\texttt{Time}_{16}/\texttt{Time}_{24} = 48.37/37 = 1.3$. The corresponding efficiency for the parallel solution is 87%. The value for the efficiency is calculated by dividing the speedup by $24/16 = 1.5$ – the relative increase in the number of processes. In Figure 5, using the appropriate plots, similar calculations can be made to compute the efficiencies for the larger FMS and Polling CTMCs. Note that these calculations are in conformance to the definitions of speedup and efficiency given in Note 5.1, except that the reference for these calculations is $\texttt{Time}_{16}$ (execution time on 16 processes) instead of $\texttt{Time}_1$.

Up until now, the discussion of Figures 4 and 5 have been deliberately restricted to a maximum of 24 processes. We now consider the performance of our parallel algorithm for up to 48 processes. Our first observation in Figure 4 (and Figure 5) is that the solution times for all the three case studies do not exhibit significant improvements for an increase in the number of processes between 24 and 48. Two likely causes for this poor performance are given as follows. First, we have mentioned earlier that the experiments reported in this paper are carried out on a processor bank consisting of a total of 24 dual-processor nodes, and that each node in the processor bank uses a single BCM5703X NIC. Essentially, both the processors within a single node must share the same card, which reduces the average communication bandwidth available to each processor by a half. Since parallel iterative solutions are typically communication intensive, reduction of this scale in the communication bandwidth must be the crucial factor to have caused the performance loss. The second reason behind the poor performance, to some extent, is that we were sharing the computing resources with other users. We do not have dedicated access to any of the machines in the processor bank. A number of processes owned by other users were being executed on the nodes when these results were collected. Increasing the number of processes to more than 24, further increases the load on the individual nodes in the processor bank. In this situation, even a single node can become a bottleneck for the whole set of the parallel processes.

[3]The largest models could only be solved using 16 or more processes due to the excessive RAM requirements (consider also that we did not have exclusive access to the nodes in the processor bank). We therefore used the execution times for 16 processes as the base reference to compute the speedups for greater number of processes. Note that we do not make any solid claims about speedups for these models; these minimum values in the figures are used solely for the explanation and graph plotting purposes.

6 Discussion

We now review the related work and give a comparison of the solution methods. We also give a classification of the parallel approaches which have been developed for the solution of large Markov models. We have known from the experimental results for the three case studies in Section 5 that the performance of an algorithm can possibly be influenced by the benchmark model used. Therefore, we will also name the case studies which were used to benchmark these solution approaches. We choose to refer to these approaches as parallel, rather than distributed, because all these approaches involve tightly coupled computing environments.

Recall from Section 1 that the state space explosion problem has led to the development of a number of solution methods for CTMC analysis. These methods are broadly classified into implicit and 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. The large amount of memory and compute power available with shared and distributed memory computers provide a natural way to address the state space explosion problem. Therefore, the solution techniques are either developed comprising a single thread or process (i.e., *serial*), or comprising multiple concurrent processes (i.e., *parallel*). Note that an out-of-core algorithm may be designed comprising two processes or threads, one for computation and the other for I/O (see e.g. [16,39]); however, since one process alone is responsible for the main computations, we consider it a serial algorithm. A parallel solution technique in itself is either *standard* (which store CTMCs in a sparse format using only the primary memory of the machine), *parallel implicit* (those based on the implicit CTMC storage), or *parallel out-of-core* (which rely on the out-of-core storage of CTMCs).

The primary memories available with the contemporary parallel computers are usually insufficient to store the ever-increasing large CTMC models. Therefore, the size of models addressed by the parallel approaches which store data structures explicitly and have only relied on the primary memory of the parallel computers is

relatively small; for these approaches, see e.g. [44,1,51]. A 2001 survey of such parallel solutions for CTMC analysis can be found in [10].

The parallel solution method for large CTMCs which utilised both the primary and secondary memories of parallel computer was introduced in [36, 37]. The solution method was based on the parallelisation of the (matrix) out-of-core approach of [16]. The author (in [36]) employed Jacobi and conjugate gradient squared (CGS) iterative methods for the CTMC solutions, and benchmarked the solutions using two case studies; the FMS, and the Courier Protocol model [63], with up to 54, and 94 million states, respectively. The solutions were reported using a total of 16 nodes (each node: 300MHz UltraSPARC, 256MB RAM, and 4GB disk) of a Fujitsu AP3000 server, connected by a dedicated high-speed network, AP-Net. The parallel (matrix) out-of-core solution of [5] further extended the size of the solvable models. The authors used the CGS method to solve FMS models with up to 54 million states, and used the Jacobi iterative method to solve FMS models with over 724 million states. For these solutions, they employed a cluster with 26 dual-processor nodes (each node: Pentium III 500MHz $\times$ 2, 512MB RAM, and 40GB disk) connected by switched fast Ethernet.

Parallelising the implicit methods for CTMC analysis have also been considered. We first mention the parallel implicit methods which use CTMC storage based on the Kronecker methods. A Kronecker-based parallel solution method for CTMC steady-state analysis is given in [8]. The authors use up to 7 nodes of a cluster, and report solutions of two CTMCs, the largest with over 8 million states. Another Kronecker-based parallel approach was reported in [21], which used the asynchronous two-stage iterative method of [23]. The authors presented solution of the FMS model ($k = 7$, 1.6 million states, see Table 2) on a LAN comprising 4 workstations. In [35], the author employed the method of *randomisation*[4] for CTMC analysis, and reported results for models with over 26 million states, using up to a total of 8 processors on shared memory machines. The first two implementations ([8, 21]) were realised using the parallel virtual machine (PVM) library; the latter [35] used POSIX threads.

The other work on the parallel implicit (or symbolic) methods is based on the MTBDDs (see Section 2.8) which we have also used in this paper. These are the parallel Gauss-Seidel solution of Markov chains on a shared memory machine [43] and on a cluster of workstations [64]. We discuss the latter. The authors benchmarked their solution method using three CTMC case studies – Polling, FMS, and Kanban systems, with up to 138, 216, and 1005 million states, respectively. For the implementation, they employed a cluster comprising a total of 32 dual-processor nodes (each node: Intel Xeon 3GHz $\times$ 2, 2GB RAM) connected by ($2 + 2$ Gbps) Myrinet.

A glance of the parallel solutions discussed above reveals that the solution of relatively larger models is attributed to the parallel out-of-core methods ([36,5]) and the parallel implicit methods ([64], and this paper). It supports our earlier observation that the primary memories of a parallel computer cannot cope with the ever-increasing size of the models for realistic systems. So, it becomes necessary to either store the underlying matrix of a model using the secondary memories of a parallel computer; or, to find an implicit, compact, representation for the model. In the following, we further discuss these four parallel methods because these have provided solutions for considerably large models.

The solutions presented in both [36] and [5] employ the indexed MSR sparse scheme (see Section 2.7) to store the matrices. The indexed MSR scheme was used because it is more compact than the standard MSR scheme; a compact representation requires less storage as well as incurs less disk I/O. The performance for these methods can be improved by using the compact MSR scheme which is 30% compact than the indexed MSR. These parallel out-of-core solutions have used the CGS method which converges faster than the basic iterative methods (e.g., Jacobi, Gauss-Seidel). However, the CGS method requires 7 iteration vectors and hence causes storage problems for large models. The iteration vectors can be kept on disk but it can affect the solution speed due to increased disk I/O.

In contrast, the parallel approaches presented in [43], [64], and in this paper, employ an implicit storage for CTMC storage. The CTMCs can be compactly stored using only the primary memories (this is equally applicable to the Kronecker-based solutions presented in [8, 21, 35]). A principal difference between the solution method of [64] and our approach is that the former takes the approach to store the whole matrix on each node, while we effectively partition the MTBDD and distribute the MTBDD partitions to the individual nodes. Our method hence is scalable to larger models compared to the approach of [64] (consider in Table 1 that the largest FMS model requires 2.6GB RAM to store the off-diagonal matrix alone).

An advantage of the parallel solution given in [64] over our solution method is that it uses the Gauss-Seidel method, which typically converges faster than Jacobi, and requires storage for only one iteration vector. However, we know from Section 4, Note 4.1, that the convergence rate for the parallel Gauss-Seidel method is likely to be worsened due to the multi-color ordering technique. This is also supported by

[4]The method of randomisation, also known as *Jensen's method*, or *uniformisation*, [34, 26, 28], is used for transient solutions of Markov chains. The method is also based on computing matrix-vector products.

the statistics reported in the paper ([64]) for Polling CTMCs. The number of iterations for a Polling CTMC ($k = 20$) are reported in the paper to have varied between 1869 and 1970. On the other hand, the number of iterations for the Jacobi solution of the same CTMC is 920 (see Table 2) – less than a half of the parallel Gauss-Seidel. Note also that the number of Gauss-Seidel iterations for the Polling CTMC with the natural matrix ordering is 36 (e.g., see [49], Page 114). However, we must also observe here that the number of parallel Gauss-Seidel iterations for the Kanban and FMS models reported in [64] are better than for the Jacobi method.

Unfortunately, a comparison of solution times and speedups for parallel implementations is not straight forward. The factors which can affect performance include: the type and speed of the processor used; the size of primary memory per node; and, more importantly, the topology, bandwidth, and latency of the communication network. For example, consider that the communication networks used by the implementations discussed in this section vary between fast Ethernet ([5]) to $2 + 2$ Gbps Myrinet ([64]). Moreover, we also find it important to consider whether the implementations are realised for dedicated or shared environment. We believe that, in future, it will become increasingly important to design parallel iterative algorithms which can tolerate and adapt to the dynamics of a shared computing environment.

A focus of this paper is to solve large models and hence we find it interesting to mention here the (serial) out-of-core approaches, which have been used to solve large Markov models. During the last ten years or so, out-of-core techniques for the analysis of large Markov chains have emerged as an effective method of combating the state space explosion problem. Deavours and Sanders [16, 17] were first to consider an out-of-core storage for CTMC matrices (1997). However, the storage for iteration vector(s) remained a hindrance for both implicit and explicit methods for CTMC analysis. In 2002, an out-of-core algorithm was presented in [39], which relaxed these vector storage limitations by using out-of-core storage for both the CTMCs and the iteration vector. Furthermore, to reduce the amount of disk I/O for greater performance, an out-of-core solution based on MTBDDs was introduced in [40], and was improved in [46], and [49]. The out-of-core approach of [49] allowed the solution of CTMCs with over 1.2 billion states on a single workstation, although with much higher solution times.

7 Conclusion

In this paper, we presented a parallel Jacobi iterative algorithm for the steady-state solution of large CTMCs. The algorithm can be used for the iterative solution of an arbitrary system of linear equations, of the form $Ax = b$. We analysed in detail the implementation of our parallel algorithm using up to 48 processes on a processor bank comprising 24 dual-processor nodes. The parallel implementation was benchmarked using three widely used CTMC case studies. The steady-state solutions for large CTMCs were obtained, and it was shown that the steady-state probabilities for a Polling CTMC with approximately 1.25 billion states can be computed in less than 8 hours using 24 processors. Note that the experiments were performed without an exclusive access to the computing resources.

The symbolic CTMC storage is central to our work. We used the two-layered, MTBDD-based, data structure to store large CTMCs. On the employed hardware, using the explicit methods, storage of this scale of CTMCs (in RAM) is simply not possible. Second, the parallelisation properties of the symbolic data structure permitted an efficient parallel solution for us. Third, by effectively decomposing the MTBDD into partitions, we were able to address solutions of even larger models, which was not possible using the earlier parallel MTBDD-based approaches.

In future, we intend to work on improving the performance of our algorithm. Our aim is to increasingly solve large systems and hence, to be able to use large-scale shared resources, we will tailor our algorithms and implementations for Grid environments, by increasing the adaptability, dependability, and scalability of our solution methods. Another hurdle for us in addressing larger models is that the modelling tool which we have used (PRISM) does not allow the parallel generation of models. We will consider a parallel modelling tool in future.

We also intend to work on an efficient parallelisation of the Gauss-Seidel method by overcoming the limitations of the earlier work in this context. We also plan to apply our parallel approach to other types of models, e.g., discrete time Markov chains (DTMCs), and to the other numerical problems in the areas of Markovian modelling and probabilistic model checking (e.g., computing transient probabilities for Markov chains). Moreover, we would like to consider modelling and analysis of more interesting case studies.

Finally: we are developing a distributed tool for debugging large-scale parallel and distributed systems; see [50]. Another purpose of the parallel solution presented in this paper is to use it as a test application for the debugging tool. Our intention is to develop and collect test case libraries for different applications, and to use these as templates with our distributed debugging tool. We also expect more interesting outcomes from a combination of our modelling and debugging projects.

Acknowledgement

This work is supported by the EPSRC "Pervasive Debugging" Grant GR/S63113/01 and by an Eclipse Innovation Grant from IBM. A part of the work presented in this paper was carried out when the first author was affiliated with the School of Computer Science, University of Birmingham; the author is grateful to the institute for providing funding for the work. Finally, we acknowledge PRISM developers team for providing us access to the tool.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] P. Buchholz, M. Fischer, and P. Kemper. Distributed Steady State Analysis Using Kronecker Algebra. In *Proc. 3rd International Workshop on the Numerical Solution of Markov Chains (NSMC'99)*, pages 76–95, Zaragoza, Spain, 1999.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] M. Fischer and P. Kemper. Distributed numerical Markov chain analysis. In Y. Cotronis and J. Dongarra, editors, *Proc. 8th Euro PVM/MPI 2001*, volume 2131 of *LNCS*, pages 272–279, Santorini (Thera) Island, Greece, September 2001.

[22] The Message Passing Interface Forum. Mpi: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[23] Andreas Frommer and Daniel B. Szyld. Asynchronous two-stage iterative methods. *Numer. Math.*, 69(2):141–153, 1994.

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

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

[26] W. Grassmann. Transient Solutions in Markovian Queueing Systems. *Computers and Operations Research*, 4:47–53, 1977.

[27] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[28] D. Gross and D.R. Miller. The Randomization Technique as a Modelling Tool and Solution Procedure for Transient Markov Processes. *Operations Research*, 32:343–361, 1984.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] A. Jensen. Markoff chains as an aid in the study of Markoff processes. *Skand. Aktuarietiedskr.*, pages 36, 87–91, 1953.

[35] P. Kemper. Parallel randomization for large structured Markov chains. In *Proc. the 2002 International Conference on Dependable Systems and Networks (DSN/IPDS)*, pages 657–666, Washington, DC, USA, June 2002. IEEE CS Press.

[36] 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.

[37] 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.

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

[39] 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*.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] Rashid Mehmood. Out-of-Core and Parallel Iterative Solutions for Large Markov Chains. Phd progress report 3, School of Computer Science, University of Birmingham, UK, October 2001.

[49] Rashid Mehmood. *Disk-based techniques for efficient solution of large Markov chains.* Ph.D. dissertation, Computer Science, University of Birmingham, UK, October 2004.

[50] Rashid Mehmood, Jon Crowcroft, Steven Hand, and Steven Smith. Grid-Level Computing Needs Pervasive Debugging. In *Proc. Grid 2005, 6th IEEE/ACM International Workshop on Grid Computing, Seattle, Washington, USA*, November 2005. To appear.

[51] 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.

[52] 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.

[53] 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.

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

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

[56] 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.

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

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

[59] 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.

[60] 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.

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

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

[63] C. M. Woodside and Y. Li. Performance Petri Net Analysis of Communications Protocol Software by Delay-Equivalent Aggregation. In *Proc. PNPM'91*, pages 64–73. IEEE Comp. Soc. Press, December 1991.

[64] Y. Zhang, D. Parker, and M. Kwiatkowska. A wavefront parallelisation of CTMC solution using MTBDDs. In *Proc. International Conference on Dependable Systems and Networks (DSN'05)*, pages 732–742. IEEE Computer Society Press, 2005.