A Wavefront Parallelisation of CTMC Solution using MTBDDs

Yi Zhang, David Parker, Marta Kwiatkowska
University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK
Email: {yxz, dxp, mzk}@cs.bham.ac.uk
Fax: +44 121 414 4281

Abstract

In this paper, we present a parallel implementation for the steady-state analysis of continuous-time Markov chains (CTMCs). This analysis is performed via solution of a linear equation system, which is carried out using the Gauss-Seidel iterative method. We apply wavefront techniques, which are used to create an efficient parallel execution schedule based on dependencies between subtasks. Our implementation uses symbolic data structures – multi-terminal binary decision diagrams (MTBDDs) – which provide a compact representation for large, structured CTMCs. MTBDDs prove to be very well suited to this application; firstly, by providing a significant reduction in inter-processor communication; and secondly, by allowing easy access to task dependency information. We demonstrate the effectiveness of our technique by presenting experimental results from a cluster of 32 nodes which exhibit speedups of between 9.7 and 16.5, comparable with existing parallelisations of similar CTMC analysis techniques. Thanks to the low space complexity and good convergence rate of the Gauss-Seidel method, our implementation represents an excellent candidate for parallel steady-state solution of CTMCs.

1 Introduction

Continuous-time Markov chains (CTMCs) are a commonly used model in the field of performance and dependability analysis of computer and communication systems. A variety of useful performance measures of a CTMC model can be derived from its steady-state probability distribution. This distribution can be obtained from the solution of a linear equation system, whose size is proportional to that of the model. Since CTMCs are in practice extremely large, developing efficient implementations of this solution process is an important direction of research. In this paper, we combine two approaches in this area. Firstly, we work with *symbolic* techniques, using data structures based on multi-terminal binary decision diagrams (MTBDDs) to provide a compact representation of CTMCs by exploiting high-level structure and regularity. Secondly, we develop a *parallel* implementation, where storage costs and computation time can be distributed between a number of processors.

We use the Gauss-Seidel iterative solution method to compute steady-state probabilities. In comparison to alternative iterative methods, this is an attractive option both in terms of solution time (i.e. speed of convergence) and memory consumption (storage of solution vectors). The Gauss-Seidel method is inherently sequential, making its parallelisation non-trivial. Fortunately, when matrices are sparse, as is usually the case for CTMCs, it is possible to extract parallelism from the pattern of non-empty blocks in the matrices. To achieve this, we use *wavefront* techniques, which schedule computation based on the dependencies between large numbers of small tasks.

In this paper, we illustrate that MTBDDs are in fact well suited to this approach. Firstly, they facilitate extraction of the dependency information required for the parallel implementation. Secondly, due to their compact nature, we are able to store the entire matrix on each parallel node, eradicating the usual need to exchange blocks of the matrix during solution and hence considerably reducing costly communication between nodes. We present experimental results on a set of benchmark models which demonstrate both an increase in the size of models for which solution is tractable, and, for 32 parallel nodes, speedups of between 9.7 and 16.5.

The remainder of this paper is structured as follows. Section 2 covers relevant background material: steady-state solution of CTMCs, symbolic methods, and wavefront techniques. In Section 3, we describe our parallelised symbolic implementation of CTMC solution. In Sections 4 and 5, respectively, we present experiment results to illustrate the efficiency of our approach and then compare it to related work. Section 6 concludes the paper.

2 Background

2.1 Numerical solution of CTMCs

A *continuous-time Markov chain* (CTMC) comprises a set of states S, representing all the possible configurations of the system being modelled, and a generator matrix $\mathbf{Q}$. Each non-diagonal matrix element $\mathbf{Q}_{ij}$ defines a rate for the pair of states $i, j \in S$. Each diagonal element $\mathbf{Q}_{ii}$ is defined as $-\sum_{j \neq i} \mathbf{Q}_{ij}$. This information gives both the likelihood of moving from each state to each other state, and the amount of time spent in each state. From state i, the probability that a transition to state j will occur is $\mathbf{Q}_{ij}/|\mathbf{Q}_{ii}|$. The time spent in state i before this transition occurs is modelled as a negative exponential distribution: the probability of exit occurring by time t is $1 - e^{\mathbf{Q}_{ii}t}$.

The *steady-state probability distribution* of a CTMC is a vector $\boldsymbol{\pi}$, each element $\boldsymbol{\pi}_i$ of which gives the probability of being in state i in the long run. For a large class of CTMCs, $\boldsymbol{\pi}$ can be obtained by solving the linear equation system $\boldsymbol{\pi}\mathbf{Q} = 0$ with the additional constraint that $\sum_i \boldsymbol{\pi}_i = 1$.

We consider the more general problem of solving the linear equation system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{b}$ is vector of length n. When the matrix $\mathbf{A}$ is large and sparse, as is typically the case in a CTMC setting, it is preferable to solve the linear equation system using iterative numerical solution methods, which repeatedly compute an approximation to the solution vector $\mathbf{x}$, terminating when some agreed convergence criterion has been met.

Two common options are the Jacobi and Gauss-Seidel methods. Of these, the latter is preferable since it has lower memory requirements (only one copy of the solution vector $\mathbf{x}$ must be stored; two are needed for Jacobi) and it generally converges faster. Faster still are Krylov methods such as Conjugate Gradient Squared (see e.g. [20]). These, however, usually require storage of additional vectors, which becomes infeasible for large systems. In this paper, we focus on Gauss-Seidel. The algorithm in Figure 1 shows the numerical computation performed for a single iteration of this method.

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

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

Notice how, in the ith step of the algorithm, the ith element of $\mathbf{x}$ is updated using the current (i.e. most up-to-date) values for other elements of the vector. This is the reason why only one vector need be stored and also why it exhibits faster convergence than, say, the Jacobi method. It also means, though, that the algorithm is inherently sequential, since, in step i, it uses results from the previous $i-1$ steps of the same Gauss-Seidel iteration. This can be relaxed slightly by using a block-based formulation of the algorithm (see e.g. [16]) as follows.

We assume that the matrix $\mathbf{A}$ is divided into $N \times N$ blocks, the (p, q)th block of which is denoted $\mathbf{A}_{(pq)}$, and that vectors are partitioned into subvectors of matching sizes. The pth block of a vector $\mathbf{x}$ is denoted $\mathbf{x}_{(p)}$. The (i, j)th element of submatrix $\mathbf{A}_{(pq)}$ and the ith element of subvector $\mathbf{x}_{(p)}$ are written $\mathbf{A}_{(pq)ij}$ and $\mathbf{x}_{(p)i}$, respectively. The size of matrix block $\mathbf{A}_{(pq)}$ is $n_p \times n_q$ and hence the size of $\mathbf{x}_{(p)}$ is n_p.

Using this notation, a block-based version of an iteration of Gauss-Seidel is shown in Figure 2. The pth step of the outer loop computes new values for elements of the block $\mathbf{x}_{(p)}$ of the solution vector. This is achieved using an additional small vector $\mathbf{v}$ of size $\max_{0 \leq p < N}\{n_p\}$. The most important aspect of the algorithm is that it accesses the matrix one block at a time, rather than a single element at a time. We also emphasise here that this is simply a reformulation of the standard Gauss-Seidel algorithm, rather than a variant such as block Gauss-Seidel (see e.g. [20]).

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

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

2.2 Symbolic techniques for CTMC solution

In practice, CTMCs are often extremely large. Fortunately, though, they are also often constructed from descriptions in high-level modelling formalisms and are thus inherently structured. *Symbolic* techniques for the construction, representation and analysis of CTMCs are those which exploit this regularity to produce a very compact storage mechanism, typically using data structures based on binary decision diagrams (BDDs). Symbolic approaches also exhibit other advantages, such as fast model construction and computation of reachable states. A range of data structures for CTMC representation have been developed in this area, including multi-terminal BDDs (MTBDDs) [7, 1], matrix diagrams [4] and Kronecker representations [19]. In this paper we use MTBDDs.

An MTBDD is a reduced directed acyclic graph which represents a function mapping vectors of Boolean values to real numbers. By encoding the row and column indices of a real-valued matrix into Boolean variables, we can represent the matrix as an MTBDD. As well as often providing a very compact representation of large, regular matrices, an MTBDD has the additional advantage that, being an inherently recursive data structure, it provides a convenient block decomposition of the matrix into submatrices.

Figure 3 illustrates this idea. It shows the structure of an example MTBDD (for clarity, low-level details are omitted) which represents some matrix $\mathbf{A}$. Each node of the MTBDD represents a particular submatrix of $\mathbf{A}$ and the two downward outgoing edges of that node represent a division of its submatrix into two further submatrices. For example, the topmost (root) node of the data structure represents the whole matrix $\mathbf{A}$. Descending one level, the two child nodes of the root node represent the top and bottom halves, $\mathbf{A}_0$ and $\mathbf{A}_1$, of $\mathbf{A}$, as illustrated in the figure. Descending one level further to the nodes on the third level of the MTBDD divides $\mathbf{A}_0$ and $\mathbf{A}_1$ into left and right portions, representing a division of the whole matrix $\mathbf{A}$ into quadrants. Notice that in this example the bottom right quadrant of $\mathbf{A}$ is empty and hence the second outgoing edge of the node representing $\mathbf{A}_1$ is omitted.

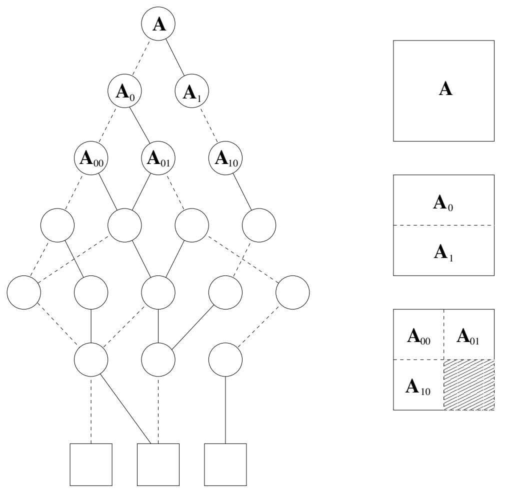


Figure 3. MTBDD representing a matrix A and its decomposition into submatrices

In [16, 18], it is shown how the MTBDD-based storage of transition matrices can be optimised by adding an explicit representation (using sparse matrix data structures) of both the high-level block structure of the matrix and the matrix blocks themselves. This improves access speed, yet maintains sufficiently compact storage.

More importantly, since it provides fast access to each row of matrix blocks and to each row of matrix entries within these blocks (which ordinary MTBDDs do not provide), it is well suited to the numerical solution of CTMCs using the block-based Gauss-Seidel method described in the previous section. The two main limiting factors which remain in this implementation are, firstly, the storage of solution vectors (proportional to the size of the CTMC matrix) and, secondly, the time for executing each iteration (proportional to the number of entries in the CTMC matrix). In this paper, we address both of these issues by distributing the problem over multiple parallel processors.

2.3 Wavefront techniques

Wavefront techniques are an approach to parallel programming that involve the division of a computation into many small tasks and the formation of an *execution schedule* for these tasks. This schedule is based on the notion of wavefronts of parallel execution. Each wavefront comprises a set of tasks which are *algorithmically independent* of each other, meaning that the correctness of the overall computation is not affected by the order in which they are performed and that they can hence be carried out in parallel. The computation order of tasks in different wavefronts, on the other hand, can affect the correctness of the overall process. The execution schedule is generated in such a way that the correctness is guaranteed. Wavefront techniques have been applied to a range of different problems, including iterative solvers (M/ILU preconditioning) [11] and particle physics simulations [13].

3 A wavefront parallelisation of Gauss-Seidel

In this section, we describe an algorithm for implementing the Gauss-Seidel method using wavefront techniques. In particular, we apply this approach to the block-based formulation of Gauss-Seidel, which means that it is suitable for implementation using symbolic techniques based on MTBDDs.

3.1 Extracting dependency information

Recall the block-based formulation for an iteration of the Gauss-Seidel method given in Figure 2. The algorithm is split into N steps, each calculating a different block of the solution vector $\mathbf{x}$. Our approach will be to distribute these steps over a number of parallel processors. Within a single iteration, the computation of the pth block $\mathbf{x}_{(p)}$ depends on the values of entries in other blocks of the vector $\mathbf{x}$. In fact, it depends on all blocks $\mathbf{x}_{(q)}$ for which $\mathbf{A}_{(pq)}$ is non-empty. Fortunately, the matrices which arise in the solution of CTMCs are typically very sparse, so there are many empty blocks in $\mathbf{A}$.

We formalise this dependency information as follows. We will call the computation of each vector block $\mathbf{x}_{(p)}$ a *task*, denoted t_p. The set of all such tasks for an iteration of Gauss-Seidel, is denoted T, i.e. $T = \{t_p : 0 \leq p < N\}$. The set of tasks upon which task t_p is dependent is $D_p = \{t_q \in T : q \neq p$ and $\mathbf{A}_{(pq)}$ is non-empty$\}$. We can then construct a *computation dependency graph*, a directed graph $G = (T, E)$ whose vertices are the tasks in T and whose edges $E \subseteq T \times T$ represent dependencies between tasks, i.e. $(t_p, t_q) \in E$ if and only if $t_q \in D_p$. As an example, Figure 4 shows the block structure of an example matrix $\mathbf{A}$ ($N = 12$) and its corresponding dependency graph.

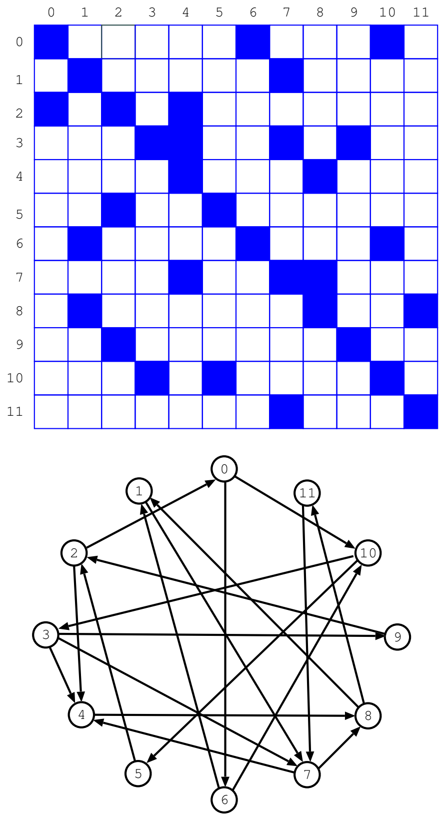


Figure 4. Block structure of an example matrix and its dependency graph

MTBDDs, which we use to store matrices, are a recursive, tree-like data structure. Extracting the nodes at a given level of this tree provides a fast and convenient representation of the block structure of a matrix (see e.g. [16, 18]). This means that the dependency information described above can be generated quickly and easily by traversing just the top layers of the MTBDD. This is illustrated by our example of Section 2.2 (see Figure 3), in which exploring the top three levels of the data structure reveals a decomposition of the corresponding matrix into quadrants. By varying the number of levels of the MTBDD which are explored, it is easy to increase or decrease the granularity of the obtained block structure.

3.2 Constructing a wavefront execution schedule

A complication of parallelising the Gauss-Seidel method is that, when computing vector block $\mathbf{x}_{(p)}$, we must use up-to-date values (i.e. computed in the current iteration) for blocks $\mathbf{x}_{(q)}$ with $q < p$ and values from the previous iteration for those with $q > p$. It is this inherently sequential nature of Gauss-Seidel which makes its parallelisation non-trivial.

The block-based Gauss-Seidel algorithm computes each vector block $\mathbf{x}_{(p)}$ in the order $p = 0, \ldots, N-1$. In fact, we can safely perform a reordering of this sequence, and hence compute the blocks in an entirely different order. Effectively this is just a permutation of the ordering of elements in the matrix $\mathbf{A}$ and vectors $\mathbf{x}$ and $\mathbf{b}$ in the original linear equation system $\mathbf{Ax} = \mathbf{b}$, which has no effect on its solution. We can exploit this fact and select an ordering which better suits our parallel implementation.

We begin by assigning a colour $c(t_p)$ to each task $t_p \in T$ in such a way that $c(t_p) \neq c(t_q)$ when $t_q \in D_p$. This process is equivalent to establishing a colouring for the dependency graph G. An example colouring for the matrix and dependency graph of Figure 4 is shown in Figure 5. We use C to denote the total number of colours used. The smaller the value of C, the greater the degree of parallelism that we will be able to obtain.

The problem of finding a minimum colouring for a graph is known to be NP-hard. For our purposes though (as our experimental results will later show), the following simple heuristic approach suffices. We iterate through the graph vertices in order, t_0 to t_{N-1}, assigning to each one a colour which has not yet been assigned to any of its neighbours (i.e. any vertex which has an edge either to or from it).

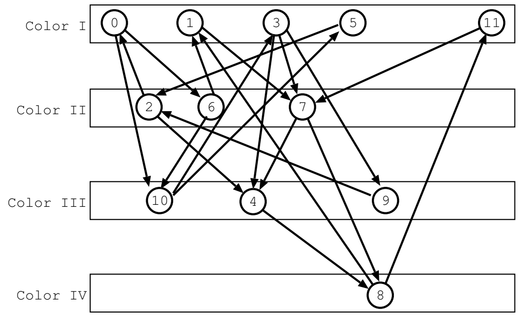


Figure 5. An example colouring for the matrix and dependency graph of Figure 4

Next, we order the C colours used in the colouring scheme according to the number of tasks to which each has been assigned, i.e. the colour with the largest number of tasks is indexed 1, and the colour with the smallest number is indexed C. In our parallel implementation of Gauss-Seidel, the order in which tasks are performed (i.e. the order in which vector blocks are computed) will be according to the ordering of their associated colours: we begin with those of colour 1 and proceed through to those of colour C. Hence the ordering on colours determines the permutation of the Gauss-Seidel ordering referred to above. Note that the ordering of tasks labelled with the same colour can be arbitrary since, by the definition of the colouring scheme, their executions are independent of each other. The fact that we place colours with more tasks earlier in the ordering is motivated by the fact that it is desirable to eliminate dependencies as early as possible.

Lastly, based on the colouring information obtained in the previous step and the number of processors available for parallel execution, we generate a wavefront execution schedule which can exploit parallelism in the implementation of Gauss-Seidel. We assume that there are P processors available, $proc_1, \ldots, proc_P$. A wavefront execution schedule is an assignment of each computation task in $T = \{t_0, \ldots, t_{N-1}\}$ to one of the P processors.

Figure 6 shows the algorithm that we use to generate such a schedule. By the definition of the Gauss-Seidel algorithm, within a single iteration, the computation of blocks of colour col will depend on the computation results of blocks of colour $1, \ldots, col{-}1$ from the same iteration. Tasks of the same colour, however, can be performed in parallel. To generate the wavefront execution schedule, we iterate through the set of tasks, ordered according to their colouring and assign them to processors in a cyclic fashion. This is done by the main loop in lines 4–14 of Figure 6: col is the colour of tasks currently being assigned, $proc$ is the current processor, T_{queue} is the set of tasks ready to be allocated to a processor, and $T_{remaining}$ is the set of tasks still to be allocated. The set T_{queue} is in fact a FIFO queue, i.e. the order in which we have inserted tasks into the queue is preserved.

When the Gauss-Seidel algorithm is executed, tasks of colour $col+1$ can actually be performed as soon as all tasks of colour $1, \ldots, col$ on which they depend have been completed. Hence, in our schedule generation algorithm, we use this information to decide the order in which tasks of the same colour are allocated a place in the schedule. More specifically, while the algorithm is assigning tasks of colour col to processors, tasks of colour $col + 1$ are appended to the queue T_{queue} as soon as all jobs of colour col on which they depend have been scheduled (and hence removed from T_{queue}). This step is performed at lines 9–12 of Figure 6. Note that no tasks of $col + 1$ are added to the queue T_{queue} until all tasks of colour col have either been assigned to a processor or are already in T_{queue}. Hence, the overall order in which tasks are allocated to processors, with respect to colour, is preserved.

Later in the paper, in Section 3.4, we will discuss how this algorithm can be improved further to optimise load balancing between processors.

```
 1. $col := 1, proc := 1$
 2. $T_{queue} := \{t_p \in T \mid c(t_p) = 1\}$ // task queue (FIFO)
 3. $T_{remaining} := T \setminus T_{queue}$ // remaining tasks
 4. do
 5.     remove a task $t_p$ from $T_{queue}$
 6.     assign $t_p$ to processor $proc$
 7.     choose next processor: $proc := (proc \bmod P) + 1$
 8.     if $(col < C)$
 9.         $T_{col+1} := \{t_p \in T_{remaining} \mid c(t_p) = col + 1\}$
10.         $T_{next} := \{t_p \in T_{col+1} \mid \forall t_q \in T_{queue}\ t_q \notin D_p\}$
11.         $T_{queue} := T_{queue} \cup T_{next}$
12.         $T_{remaining} := T_{remaining} \setminus T_{next}$
13.         $col := \min\{c(t_p) \mid t_p \in T_{queue}\}$
14. until $(T_{queue} = \emptyset)$
```

Figure 6. Algorithm to generate a wavefront execution schedule.

3.3 Parallelisation

In this section, we describe a parallelisation of the Gauss-Seidel method, based on the wavefront execution schedule described in the previous section. The first step is for one of the processors to generate this schedule and distribute it to all other processors. Subsequently, each processor $proc_i$, where $i = 1, \ldots, P$, has been assigned a set of tasks T_i. In each iteration of Gauss-Seidel, processor $proc_i$ will be responsible for the computation and storage of every vector block $\mathbf{x}_{(p)}$ for which $t_p \in T_i$. The algorithm executed by processor i on iteration k of Gauss-Seidel is illustrated by the pseudo-code in Figure 7.

Each processor works through the C colours in order, processing each task t_p of that colour which has been assigned to it. For each task t_p, it computes the vector block $\mathbf{x}_{(p)}$. The computations required for Gauss-Seidel (see Figure 2) correspond to lines 3, 9, 11, 14 and 15 of Figure 7. These computations require access to vector blocks $\mathbf{x}_{(q)}$ for which $t_q \in D_p$. In many cases, the values for these blocks must be obtained from other processors. To achieve this, remote requests are sent in lines 5–8 and then responses are received in line 10. In fact, the order in which blocks are used (in the for loop at line 9) is not important so, for efficiency, blocks can be used in the order in which they become available. Conversely, remote requests from other processors must also be responded to. This is done at each

```
1. for $(1 \leq col \leq C)$
2.   for each $t_p \in T_i$ with $c(t_p) = col$
3.     $\mathbf{v} := \mathbf{b}_{(p)}$
4.     for each $t_q \in D_p$ with $t_q \notin T_i$
5.       if $(c(t_q) > col)$
6.         send remote request for values of $\mathbf{x}_{(q)}$ at iteration $k-1$
7.       else
8.         send remote request for values of $\mathbf{x}_{(q)}$ at iteration $k$
9.     for each $t_q \in D_p$
10.      if $(t_q \notin T_i)$ wait for requested values of $\mathbf{x}_{(q)}$
11.      $\mathbf{v} := \mathbf{v} - \mathbf{A}_{(pq)}\mathbf{x}_{(q)}$
12.      respond to any remote requests
13.    ensure all remote requests for $\mathbf{x}_{(p)}$ at iteration $k-1$ responded to
14.    for $(0 \leq i < n_p, i \neq j)$
15.      $\mathbf{x}_{(p)i} := (\mathbf{v}_i - \sum_{0 \leq j < n_p} \mathbf{A}_{(pp)ij} \cdot \mathbf{x}_{(p)j}) / \mathbf{A}_{(pp)ii}$
16.    respond to any remote requests
```

Figure 7. Algorithm for processor $proc_i$ on iteration k of Gauss-Seidel.

convenient point in the algorithm (lines 12 and 16).

In order to ensure that our parallel algorithm of Figure 7 represents a true implementation of the Gauss-Seidel method, we require that the correct values of each vector block $\mathbf{x}_{(q)}$ are used in line 11. The ordering for Gauss-Seidel is determined by the colouring scheme. Hence, in the case where $c(t_q) < c(t_p)$, the values should be from the previous, i.e. $(k-1)$th, iteration, and in the case where $c(t_q) > c(t_p)$ the values should be from the current, i.e. kth, iteration. The requests sent in lines 5–8 and the wait command in line 10 ensure that this is satisfied. In line 13, the processor verifies that all remote requests for values (from the previous iteration) of its current block have already been received and responded to. Only then does it proceed and update the block to its new values. This ensures that the values are not overwritten before they are needed elsewhere. This verification process is possible because each processor has access to both the block structure of the matrix $\mathbf{A}$ (in fact the whole matrix) and to the overall execution schedule.

We have already described, in Section 3.1, how the MTBDD-based data structure that we use to store the matrix $\mathbf{A}$ facilitates extraction of the dependency information used to construct the colouring scheme. The second advantage of using MTBDDs for the parallel implementation now also becomes clear. Note that all nodes will require access to multiple blocks of $\mathbf{A}$. With a conventional, explicit storage scheme, these blocks would usually need to be communicated to the processors as required. With our approach, the matrix representation is typically compact enough that the whole matrix can be stored locally on each processor and its blocks can be quickly and conveniently accessed without any communication overhead. It should be noted that this approach is equally applicable to alternative symbolic matrix storage schemes, such as Kronecker representations.

3.4 Implementation optimisations

Our parallelisation of Gauss-Seidel is targeted at PC clusters connected through Ethernet or Myrinet [3]. In our algorithm, communication between processors is implemented using MPI (Message Passing Interface) [17] and, in particular, the MPICH [9] implementation. We have employed several optimisation techniques in this respect: computation and communication interleaving, load-balancing and remote block caching.

The parallel Gauss-Seidel method requires a substantial amount of communication between processors: blocks of the solution vector must be passed between processors many times during each iteration. As the size of CTMC being solved increases, so does the amount of communication required. The impact of this overhead on the performance of the algorithm can be lessened considerably by interleaving operations for the communication and computation of vector blocks. For example, in lines 10–12 of Figure 7, computation using a particular vector block $\mathbf{x}_{(q)}$ can be performed concurrently with the sending and receiving of blocks other than $\mathbf{x}_{(q)}$. To implement this idea, we use non-blocking MPI functions (MPI_Iprobe, MPI_Isend and MPI_Irecv) for inter-processor communication.

Another important factor for the performance of parallel programming is load balancing, which aims to distribute computation evenly between processors and minimise communication load between them. In our approach, each node is allocated a set of vector blocks. It is then responsible for both the computation of these blocks and the sending of computation results to other nodes which need them. We gave an algorithm to perform this allocation in Figure 6, in

Table 1. Model parameters and statistics.

Model	States	Transitions	Blocks	Size (MB)	
			(N)	MTBDD	Sparse
FMS (K=11)	54,682,992	518,030,370	1,365	297	6,137
FMS (K=12)	111,414,940	1,078,917,632	1,820	558	12,772
FMS (K=13)	216,427,680	2,136,215,172	2,380	1,005	25,273
Kanban (K=7)	41,644,800	450,455,040	120	18	5,314
Kanban (K=8)	133,865,325	1,507,898,700	165	43	17,767
Kanban (K=9)	384,392,800	4,474,555,800	220	95	52,674
Kanban (K=10)	1,005,927,208	12,032,229,352	286	195	141,535
Polling (K=20)	31,457,280	340,787,200	308	65	4,020
Polling (K=21)	66,060,288	748,683,264	324	141	8,820
Polling (K=22)	138,412,032	1,637,875,712	340	307	19,272

which tasks (i.e. blocks) are removed from a waiting list and assigned to processors in a round-robin fashion. We have improved this algorithm as follows. Each processor is associated with two cost values, one for computation and one for communication. Whenever a block is assigned to a processor, the computation cost of that processor is increased. The communication cost for both this processor and all others which need the block are also increased. When a processor is allocated a task (line 6 of Figure 6), preference is given to those with the smallest communication cost and, in the case of a tie, to those with the smallest computation cost. More sophisticated load balancing methods can also be incorporated into the allocation algorithm but we do not consider these here.

Even with the above optimisation techniques in place, the parallel Gauss-Seidel method is still communication intensive in some instances. To further reduce communication between processors, we apply caching techniques to the sending and receiving of blocks between processors. Since each processor is typically responsible for more than one block, the computation of these blocks may require remote retrieval of the same block more than once. Hence, each processor counts the number of times it will use each block and, where possible, caches them until they are no longer required. The experimental results presented in the next part of the paper make use of all three optimisation techniques described here.

4 Experimental results

In this section, we present experimental results for our wavefront parallelisation of Gauss-Seidel, which has been implemented as part of the PRISM model checking tool [15]. We used three benchmark CTMC models: a flexible manufacturing system (FMS) [6], a Kanban manufacturing system [5] and a cyclic server polling system [10]. For each, we generated CTMCs of several sizes by varying a parameter (K) of the model (see www.cs.bham.ac.uk/~dxp/prism for more information). We computed the steady-state probability distribution, terminating Gauss-Seidel when the maximum relative difference between solution vector values reached 10^{-6}, i.e. when:

$$\max_{0 \le i < n} \left(\frac{|\mathbf{x}_i - \tilde{\mathbf{x}}_i|}{|\mathbf{x}_i|} \right) < 10^{-6}$$

where $\mathbf{x}$ and $\tilde{\mathbf{x}}$ denote the solution vector from the current and previous iteration, respectively.

Table 1 shows statistics for each CTMC: the size (both the number of states and the number of transitions) and the number of blocks we partition the solution vector into (i.e. N). For the latter, we select a partition which minimises storage requirements (see [16]). Table 1 also includes the amount of memory required to store the matrix representing each CTMC, using both our MTBDD-based data structure and using a standard sparse matrix storage scheme. Notice that the MTBDD representation is: (a) much smaller – in the worst case (FMS, K=11), still a factor of 20 smaller; and (b) always less than 1 GB – compact enough to fit into memory on our setup.

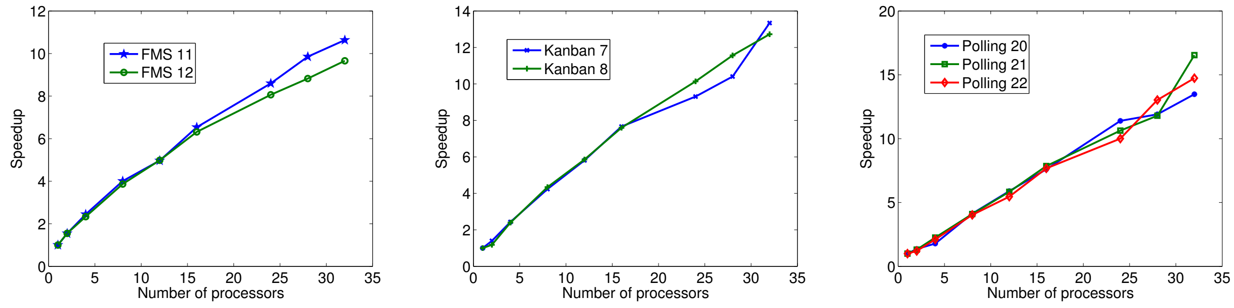
Our experiments were performed on a Myrinet-connected PC cluster with 50 nodes, each equipped with dual 3 GHz Intel P4 Xeon processors and 2 GB of RAM. In fact, we had exclusive access to just 32 of the nodes and used only one processor on each.

Table 2 shows the total execution time (seconds) for experiments on various numbers of nodes. Experiments which could not be completed due to excessive memory requirements are marked “O/M”.

Our first observation is that, for three examples (FMS K=13, Kanban K=9, 10), our parallel implementation allowed analysis of CTMCs which was intractable in the serial case (i.e. 1 node). Solution for these two examples was

Table 2. Total solution time (seconds) for the wavefront parallel Gauss-Seidel method.

Num.	FMS			Kanban				Polling		
nodes	K=11	K=12	K=13	K=7	K=8	K=9	K=10	K=20	K=21	K=22
1	15,990	35,637	O/M	4,683	19,417	O/M	O/M	8,764	14,195	45,485
2	10,349	22,986	O/M	3,351	16,419	O/M	O/M	6,451	10,834	37,713
4	6,548	15,264	O/M	1,925	8,099	34,755	O/M	4,906	6,301	21,553
8	3,991	9,212	O/M	1,106	4,474	16,271	O/M	2,123	3,463	11,287
12	3,218	7,148	O/M	806	3,314	11,452	45,206	1,488	2,433	8,338
16	2,446	5,642	12,544	611	2,555	9,522	29,674	1,153	1,807	5,929
24	1,860	4,419	9,657	503	1,915	6,741	20,560	769	1,335	4,546
28	1,623	4,038	8,173	450	1,679	5,753	18,599	736	1,203	3,491
32	1,504	3,689	7,693	351	1,526	5,134	15,750	650	858	3,086



(a) FMS (b) Kanban (c) Polling

Figure 8. Speedup of the wavefront parallel Gauss-Seidel method

not possible for less than 16, 4 and 12 nodes, respectively. Note that the FMS model, although smaller in terms of the number of states, requires more memory for matrix storage, and thus more parallel nodes to solve (the fewer the number of nodes, the higher the amount of RAM required for vector block storage on each node).

Secondly, we see that the parallelised versions have successfully produced a significant reduction in total run-time. For the cases where we were able to execute the sequential version, the speedup when using all 32 nodes ranges between 9.7 and 16.5. In Figure 8, we plot the speedup obtained in these cases for each number of parallel nodes. For the models where sequential results are not available, we can still judge the performance of the parallel implementation by examining relative speedup. For all three FMS models, the relative speedup between 16 and 32 processors is quite consistent, ranging between 1.53 and 1.63. Similarly, for the Kanban models, the relative speedup between 12 and 32 ranges between 2.17 and 2.87. We can hence expect the absolute speedup (i.e. between 1 node and 32 nodes) to be comparable for the larger models.

We conclude our analysis by discussing the convergence of our Gauss-Seidel algorithm. As discussed in Section 3.2, the allocation of vector blocks between parallel processors effectively corresponds to a permutation of the matrix $\mathbf{A}$ of the linear equation system being solved. This means that the convergence of Gauss-Seidel may vary between experiments for the same system but on different number of nodes. Table 3 shows the number of iterations required for convergence on different numbers of parallel nodes for one CTMC from each case study. Only a small variation in convergence is observed. Similar results were obtained on the other CTMCs used in this paper. Overall, the maximum variation in convergence observed for the case studies used here was 7.4%.

5 Related work

Since the sequential nature of the Gauss-Seidel method makes its parallelisation difficult, the parallel implementations which have been developed are usually application-specific. Examples include solution of partial differential

Table 3. Number of iterations for the wavefront parallel Gauss-Seidel method.

Num. nodes	FMS K=11	Kanban K=7	Polling K=20
1	1481	763	1869
2	1481	763	1871
4	1480	760	1876
8	1490	765	1884
12	1501	759	1890
16	1511	759	1866
24	1511	762	1804
28	1509	759	1920
32	1501	757	1970

equations [8] and of equation systems from electrical power systems applications [14]. In the latter case, matrices are permuted into a block-diagonal-bordered form and a graph multi-colouring scheme is used to identify available parallelism. This approach is not applicable in our setting since matrix permutation for MTBDDs is expensive. They used a multi-colouring scheme with which each block may have several colours and achieved relative speedups of between 8 to 12 for 32 processors.

Progress has also been made in the parallelisation of iterative steady-state solution of CTMCs [2, 12]. In these cases, though, solution is not performed using the Gauss-Seidel method, but the Jacobi and Conjugate Gradient Squared (CGS) methods. These alternatives have the advantage that they are less sequential in nature and hence more amenable to parallelisation, but as observed in Section 2.1, have their own disadvantages: Jacobi is slower to converge and requires two solution vectors; CGS is generally faster to converge but uses at least six vectors. In [2], the authors work on a cluster of PCs, like in this paper. On 26 nodes, they achieve speedups of 8 and 18 for Jacobi and CGS, respectively. In the latter case, however, some of the speedup can be attributed to the additional time required for the serial version to save intermediate vectors to disk. In [12], which uses disk-based techniques, the best speedup is 6.5 on 16 processors for the CGS method. Our own earlier work in [16] also parallelises iterative steady-state solution of CTMCs but is developed for a different parallel architecture: multiple processors on the one computer, communicating via shared memory.

6 Conclusion

In this paper, we have presented a technique for the parallelisation of steady-state solution of CTMCs using the Gauss-Seidel iterative method, aimed at clusters of PCs with Ethernet or Myrinet connections. We use symbolic, MTBDD-based data structures for matrix storage and wavefront techniques to devise an efficient execution schedule. We have demonstrated how the symbolic approach is particularly well suited to our parallel algorithm, firstly because it facilitates extraction of information about dependencies between tasks to be executed in parallel, and secondly because its compactness allows the matrix to be stored on all nodes, dramatically reducing the amount of communication required. Our experimental results illustrate that our implementation is efficient, exhibiting a speedup of between 9.7 and 16.5 on 32 processors. This, combined with the advantages of the Gauss-Seidel algorithm, mean that our algorithm represents an excellent candidate for the steady-state solution of large CTMCs.

In the future, we would like to improve this work in a number of ways. Firstly, we plan to test the performance of our Gauss-Seidel algorithm on different types of analysis, including reachability-based properties for both CTMCs and DTMCs. Subsequently, we would like to consider alternative iterative solution techniques not based on Gauss-Seidel, such as transient analysis of CTMCs and, more generally, model checking of the logic CSL on CTMCs and of the logic PCTL on DTMCs or MDPs. We would also like to optimise our implementation by improving the degree of parallelism on a single parallel node, e.g. by exploiting the fact that dual processors are available.

Acknowledgements

The work in this paper was supported in part by EPSRC grants GR/S11107 and GR/S27252. The authors would also like to thanks the anonymous referees for their helpful comments.

References

[1] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[2] A. Bell and B. Haverkort. Serial and parallel out-of-core solution of linear systems arising from generalised stochastic Petri nets. In *Proc. High-Performance Computing Symposium (HPC), ASTC'01*, pages 242–247. Society for Computer Simulation, 2001.

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[4] G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz and M. Silva, editors, *Proc. 8th International Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31. IEEE Computer Society Press, 1999.

[5] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stocastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

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

[7] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10((2/3):149–169, 1997.

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

[9] 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, Sept. 1996.

[10] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[11] W. Joubert, T. Oppe, R. Janardhan, and W. Dearholt. Fully parallel global M/ILU preconditioning for 3-D structured problems. Report LA-UR-98-2259, Los Alamos National Laboratory, May 1998.

[12] W. Knottenbelt and P. Harrison. Distributed disk-based solution techniques for large Markov models. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 58–75. Prensas Universitarias de Zaragoza, 1999.

[13] K. Koch, R. Baker, and R. Alcouffe. Solution of the first-order form of threedimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, 65(198), 1992.

[14] D. Koester, S. Ranka, and G. Fox. A parallel Gauss-Seidel algorithm for sparse power system matrices. In *Proceedings of Supercomputing'94*, pages 184–193, 1994.

[15] 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. IEEE Computer Society Press, 2004.

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

[17] MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

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

[19] 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*, volume 13(2) of *Performance Evaluation Review*, pages 147–153, 1985.

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