Form Method Syst Des (2006) 29:177–196
DOI 10.1007/s10703-006-0007-0

Distributed disk-based algorithms for model checking very large Markov chains

Alexander Bell · Boudewijn R. Haverkort

Published online: 8 July 2006

Abstract In this paper we present data structures and distributed algorithms for CSL model checking-based performance and dependability evaluation. We show that all the necessary computations are composed of series or sums of matrix-vector products. We discuss sparse storage structures for the required matrices and present efficient sequential and distributed disk-based algorithms for performing these matrix-vector products. We illustrate the effectivity of our approach in a number of case studies in which continuous-time Markov chains (generated in a distributed way from stochastic Petri net specifications) with several hundreds of millions of states are solved on a workstation cluster with 26 dual-processor nodes. We show details about the memory consumption, the solution times, and the speedup. The distributed message-passing algorithms have been implemented in a tool called PARSECS, that also takes care of the distributed Markov chain generation and that can also be used for distributed CTL model checking of Petri nets.

Keywords Markov chains · Matrix-vector product · Disk-based algorithms · State-space generation · CSL model checking · Distributed algorithms

1. Introduction

Over the last decade, very efficient algorithms to generate state-spaces and state-transition relations from high-level model specifications in a parallel or distributed manner have been reported, e.g., when stochastic Petri nets (SPNs) are used as high-level description technique

A. Bell (✉)
Numerical Analysis and Computational Mechanics, Department for Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands
e-mail: a.bell@math.utwente.nl

B. R. Haverkort
Design and Analysis of Communication Systems, Department for Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands
e-mail: brh@cs.utwente.nl

(cf. [3, 10, 21, 26, 30]), allowing for the explicit generation of continuous time Markov chains (CTMCs) with hundreds of millions of states. In this paper we focus on the case where such state-spaces and transition relations are to be used as part of a model checking procedure, in particular, when used to determine system performance and dependability properties that are specified as CSL properties [4], that is, when the underlying state-transition system can be interpreted as a finite CTMC. However, the evaluation of these CTMCs also requires efficient parallel/distributed numerical algorithms; such algorithms are the topic of this paper. By now, it is widely acknowledged that the numerical evaluation of very large CTMCs is much more computationally intensive than their generation, so that a focus on the solution algorithms seems most appropriate.

In what follows, we focus on the case where indeed SPNs are used, and where continuous-time is involved. Our algorithms do, however, apply equally well when other high-level models are used, or when a discrete-time analysis is involved. Furthermore, we focus on the use of clusters of workstations, because these offer large amounts of aggregated main memory and computational power at a distinguished price-performance ratio. In doing so, we adopt a message-passing programming style for distributed memory based on the MPICH implementation [18, 19] of MPI [16].

The algorithms we present operate on data structures that are partially stored on disk (simply because the data is too big to fit in main memory); this is why we call our algorithms disk-based (or "out-of-core" algorithms). This has been done in the past for sequential solution of Markov chains as well [14]. Other researchers have also studied the parallel and distributed solution of CTMCs, cf. [25, 27–29], sometimes combining a symbolic representation of the CTMC with an explicit representation of the probability vector, sometimes using disk-based data structures. In our work, we only use explicit (sparse) data structures. For that reason, our algorithms execute relatively fast in comparison. Furthermore, our experiments have used the largest number of processors, and address the largest state spaces, so that real insight in the achievable speedup is obtained.

This paper is further structured as follows. We recall the basics of explicit generation of very large CTMCs as well as the characteristic equations for their steady-state and transient probabilities in Section 2. We continue with an overview of sparse matrix storage techniques for CTMCs, as well as some sequential algorithms for them, in Section 3. Section 4 then presents efficient distributed disk-based matrix-vector product algorithms, that are used at the core of any other solution algorithm. Section 5 presents and discusses results for one case study; in the appendix results for another case study are given. All our experiments have been performed on a cluster with up to 26 dual-processor nodes. Section 6 concludes the paper.

2. Markov chain generation and characteristic equations

In this section we give a concise review of how to generate Markov chains from high-level (SPN) models. In particular, in Section 2.1 we explain how the state-space and the transition relation are partitioned in order to find a good trade-off between communication and computation. Then, in Section 2.2, we present the equation systems that need to be solved.

2.1. State-space and transition relation generation

Throughout the rest of this paper we use (generalised) stochastic Petri nets as modelling language [1]; SPNs have proven themselves over the last 30 years as suitable formalism

to describe system performance and dependability properties [2], and have also been used lately in combination with CSL model checking.

Given an SPN, in principle, a simple depth- or breadth-first search algorithm can be used to generate all states, as e.g., reported in [20]. Key to such an algorithm is a data structure to store generated states, that can be searched quickly. Where previously binary trees where used for this purpose, lately, it has become clear that the use of hash tables is most efficient for this [21]; on a reasonable PC (1 GHz, 1024 MB), state-spaces can be generated sequentially at a rate of around 10,000 states per second. The state transition relation that is also discovered during the state-space generation algorithm, is not further needed in the generation algorithm and can therefore be stored in a simple format on disk. Since the state transitions carry rate information (due to the stochastic nature of the model), storage in a BDD is less appropriate; storage of the transition relation in an MTBDD has been considered [22, 23], but appears less suitable for our purposes (it results in slower numerical algorithms). The transition relation describes a CTMC, of which all the information can be summarised in an $n \times n$ generator matrix Q, of which the (i, j)-th entry $(i \neq j)$ describes the transition rate between state i and j. The basic generation algorithm can be adapted to run in a distributed fashion easily, by introducing a so-called partitioning function that relates a state identifier to a processor index. The partitioning function assigns work (that is, states that need to be investigated further) to processors. The generation algorithms per processor are the same as in the sequential case, except for the fact that each processor may receive work (states) from other processors, and each processor may forward work (states) to be investigated to other processors. A distributed termination detection algorithm is used to decide upon completion. After completion, the overall state space is available, partitioned over the processors, as is the transition relation.

The distributed state-space generation algorithms have been developed and implemented in the tool PARSECS (**Par**allel **S**tate-space **E**xplorer and Markov **C**hain **S**olver). PARSECS fully supports the stochastic Petri net language CSPL [9] and is used as the basis for the algorithms described in this paper. PARSECS has been written in C++ and uses the de facto standard MPI (message passing interface) [19] to facilitate the communication in the distributed computations; for more details, we refer to [21]. Recently, we have extended PARSECS to include distributed algorithms for CTL model checking [7].

Throughout the paper we will refer to a widely used benchmark model, the so-called flexible manufacturing system (FMS) [12]. It is a straightforward model of a number of interacting production lines, through which a number of pallets k with work items circulate. Table 1 presents the number of states n and the number of nonzero (off-diagonal) transitions a in the underlying CTMC.

Table 1 No. states (n) and transitions (a) for the FMS model

k	n	a	k	n	a
4	35 910	237 120	10	25 397 658	234 523 289
5	152 712	1 111 482	11	54 682 992	518 030 370
6	537 768	4 205 670	12	111 414 940	1 078 917 632
7	1 639 440	13 552 968	13	216 427 680	2 611 411 257
8	4 459 455	38 533 968	14	403 259 040	4 980 958 020
9	11 058 190	99 075 405	15	724 284 864	9 134 355 680

2.2. Characteristic equations

As described previously, once the state-space and transition relation have been generated, the complete stochastic behaviour of the model is characterised by the generator matrix Q. To compute steady-state performance or dependability measures, as well as to evaluate the CSL steady-state operator $\mathcal{S}_{\bowtie p}(\cdot)$, requires the computation of the steady-state distribution π (of length n, which can be enormous) of the CTMC through the solution of the global balance equations [31]:

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

For the computation of the steady-state solution we may divide all entries of the generator matrix Q by the negated diagonal entry of the corresponding row:

$$R = -QD^{-1} \quad \text{with} \quad D = \text{diag}\,(q_{0,0}, \ldots, q_{n-1,n-1}). \quad (2)$$

Thus, all diagonal entries of R then equal -1. Instead of solving $0 = \pi Q$ we then solve y from

$$0 = \pi DQD^{-1} = -yR \quad \text{with} \quad y = -\pi D. \quad (3)$$

The steady-state distribution π is then easily obtained as $\pi = -yD^{-1}$. Using this technique one does not need memory for the diagonal entries during the solution process (since they are all equal to -1). For large n solving (3) can only be done using iterative techniques, the most prominent being the method of Jacobi, the method of Gauss-Seidel and Krylov subspace methods (e.g. Conjugate Gradient Squared) [31]. The choice between these methods depends, as usual, on their time efficiency, however, most important is their storage efficiency. Krylov subspace methods, for instance, typically require multiple intermediate vectors (arrays of doubles of length n) to be stored; this can be prohibitive when dealing with large state-spaces. Furthermore, some methods might be less easy to distribute, even though their efficiency is known to be very good in the sequential case. This applies, for instance, to the Gauss-Seidel method that sequentially normally outperforms the Jacobi method, but suffers from data dependencies.

Irrespective the choice of method, all methods have in common that they compute successive approximations of the steady-state probability vector π through a series of matrix-vector products. That is, an initial guess to the solution is taken and stored in a vector x, a better approximation is obtained by multiplying x with some matrix A, which equals the generator Q or has been derived from it (like the matrix R above): $x' \leftarrow A \cdot x$. A is called the iteration matrix, and its characteristics (eigenvalues, among others) determine the speed of convergence. Convergence is best tested using a (relative) residual criterion, that is, by verifying whether the obtained approximation x indeed fulfils (1).

In order to compute time-dependent (transient) performance and dependability properties of a system, or to validate time-bounded until properties ($\mathcal{P}_{\bowtie p}(\phi\ \mathcal{U}^{\leq t}\ \psi)$) in CSL, we need to determine the transient state probabilities of the CTMC. These are characterised by a linear system of differential equations:

$$\pi'(t) = \pi(t) \cdot Q, \quad \text{given}\ \ \pi(0). \quad (4)$$

A technique known as uniformisation is best suitable for this. It transforms the above equation to the discrete-time domain by defining a matrix $P = I + Q/q$ (with I the identity matrix and q the largest absolute diagonal element of Q) that is used to define the series $\pi^{(i+1)} = \pi^{(i)} \cdot P$ with $\pi^{(0)} = \pi(0)$. The transient probability vector $\pi(t)$ then equals

$$\pi(t) = \sum_{i=0}^{\infty} \Psi(qt, i)\pi^{(i)}, \quad (5)$$

where $\Psi(qt, i) = e^{-qt}(qt)^i / i!$ are the so-called Poisson probabilities, and where the infinite summation can be truncated to achieve a certain preset accuracy criterion. What is important to observe is that also in this case, the core of the solution algorithm is formed by matrix-vector products. For that reason, we focus in what follows on efficient distributed algorithms for that purpose.

3. Storage schemes and sequential matrix-vector product

In this section we present sparse storage structures for the matrices and vectors we have to deal with in Section 3.1, followed by a discussion of two key matrix operations in Section 3.2.

For the rest of this section we assume 32-bit processor architecture, using 4 bytes to store an `integer`, 2 bytes for a `short` and 1 byte for a `char`. Floating point values comprise 8 byte for a `double` and 4 byte for a (single precision) `float`. We note that `floats` typically suffice for the storage of the generator matrix, although their usage decreases the achievable performance due to required type conversions as doubles must be used for solution vectors to ensure an adequate acuracy. Therefore we will not consider the usage of `floats`. We will use C-style indexed arrays, that is, an array storing m elements in indexed from $0, \ldots, m-1$.

3.1. Sparse matrix storage schemes

We denote with n the number of states of a CTMC, equalling the number of rows (and columns) of the matrix to be stored and use a as the number of off-diagonal nonzero entries of the generator matrix. Table 1 shows values for these two parameters for the FMS model.

We introduce our approach to the sparse storage of generator matrices in three steps. First, we consider only the rate matrix, ignoring the diagonal entries of the generator matrix, and recall the most fundamental sparse storage schemes. In the second step we introduce a technique called indexing that achieves substantial savings of memory. As last step we consider the storage for the diagonal entries.

3.1.1. Basic sparse storage formats

We present three storage formats for general sparse matrices and discuss their adaptability for the storage of rate matrices, i.e., generator matrices of a CTMC without diagonal entries.

Sparse (S) format, also known as *coordinate* format (cf. [15]), is mainly used as a matrix exchange format. For each nonzero entry of a matrix the position, consisting of the corresponding row and column number, is stored as an integer, while the actual value of the entry is stored as a double. This results in three arrays of size a. Using this approach we need $16 \cdot a$ bytes to store the rate matrix, hence the rate matrix of the CTMC resulting from the

FMS model with parameter $k = 5$ can be stored using 17 MB of memory. The entries may be stored in an arbitrary order, although it is common to sort them by rows or columns. The matrix given below will be used as an example throughout this section.

row	0	0	1	1	1	2	2	2	3	3	4
col	1	4	0	2	4	1	3	4	2	4	0
val	0.03	0.001	1.0	0.02	0.001	1.0	0.01	0.001	1.0	0.001	0.2

Compressed sparse row/column (CS[R\|C]) format, is a sparse matrix format used in the sparse BLAS toolkit [8]. The CSR format abandons to store the actual row number for each nonzero entry but uses an array `row_i` of length n to store an index at which position in the column and value arrays the entries of the corresponding row start. To ease the development of algorithms employing this format sometimes an array of length $n+1$ is used. In this case the column and value entries of row i can be found at positions `row_i[i]`, ..., `row_i[i + 1] − 1`. The compressed sparse column format is defined in an analogous manner. Using integers for the indices and the actual row/column numbers and doubles for the actual values this results in a total memory requirement of $12 \cdot a + 4 \cdot n$ bytes for the storage of a rate matrix. The contents of the arrays used by the CSR-format for the example rate matrix are as follows:

`row_i`	0	2	5	8	10	(11)	—	—	—	—	—
col	1	4	0	2	4	1	3	4	2	4	0
val	0.03	0.001	1.0	0.02	0.001	1.0	0.01	0.001	1.0	0.001	0.2

The storage requirements for the model FMS ($k = 5$) are reduced to 13.3 MB using this format.

Modified compressed sparse row/column format (mCS[R\|C]) exploits the fact that rate matrices are very sparse. This storage scheme was originally introduced in [24], but first named as such in [6]. For CTMCs arising from SPNs, the number of timed transition gives an upper bound on the maximum number of nonzero entries per row in the rate matrix. A `short` or even a `char` is typically sufficient to store the number of non-zeroes occurring in a certain row. Hence, in this format an array called `row_n` of `char` is employed that stores the number of nonzero entries occurring in row i at position i. The arrays used to store the column positions and the actual values are the same as with the CSR format. The resulting entries for these three array for the example rate matrix are given below:

row_n	2	3	3	2	1	—	—	—	—	—	—
col	1	4	0	2	4	1	3	4	2	4	0
val	0.03	0.001	1.0	0.02	0.001	1.0	0.01	0.001	1.0	0.001	0.2

Using this format the memory requirements for the storage of a rate matrix are lowered to $12 \cdot a + n$ and the FMS ($k = 5$) rate matrix can stored using 12.9 MB, if a character is sufficient.

3.1.2. Indexing

The formats considered so far can be improved if one considers not to store the actual nonzero entries, but indices to them. As the number of different entries occurring in the rate matrices is very small a `char` or `short` suffices to differentiate between 256 or 65536 different values.

As long as no immediate transitions or marking dependent rates are involved the number of different values is limited by the number of timed transitions. In the general case, the number of different values occurring in the rate-matrix can only be determined by generating it, hence the indexing has to be done in either a post-processing step or by assuming an upper bound. For the SPN models we encountered so far, the number of actually occurring different entries in the rate matrix is below 255 and hence, an array of `char` (called `val_i`) suffices to store the indices to the actual values, which are stored in a separate array (called values) of type `double` with length the number of different entries in the rate-matrix. The resulting allocation of the arrays using indexing in combination with the mCSR-format for the example rate-matrix is given below:

row_n	2	3	3	2	1	—	—	—	—	—	—
col	1	4	0	2	4	1	3	4	2	4	0
val_i	0	1	2	3	1	2	4	1	2	1	5
values	0.03	0.001	1.0	0.02	0.01	0.2	—	—	—	—	—

In the following we will refer to sparse storage schemes using indexing by appending "_i" to the names given previously. The resulting memory requirements for the storage of a rate matrix using `char` respectively `short` arrays for the indices are given in the table below.

Format	Char indices	short indices
S_i	$9 \cdot a$	$10 \cdot a$
CSR_i	$5 \cdot a + 4 \cdot n$	$6 \cdot a + 4 \cdot n$
mCSR_i	$5 \cdot a + n$	$6 \cdot a + n$

The above given numbers do not include the storage for the actual values, but the memory required for that is negligible compared to the overall memory requirements.

3.1.3. Storing the diagonal

We now address the storage of the diagonal entries in order to represent the entire generator matrix. Again the chosen storage variant of the diagonal is independent from the employed sparse format of the rate matrix and, hence, each of the variants discussed below may be combined with any of the formats introduced above. There are four options to store the diagonal entries:

1. We can store the diagonal as an array of `double` or `float`, resulting in memory requirements of $8 \cdot n$ respectively $4 \cdot n$. This approach simplifies the implementation as no special routines for the "diagonal vector" as, e.g., used in the method of Jacobi, are required.
2. The idea of indexing can be applied to the diagonal as well. As the diagonal entries equal the negative row-sums the number of different entries can not easily be determined without computing the generator matrix.
3. An option that requires no memory at all for the storage of the diagonal has already been mentioned in Section 2.2, but is limited to steady-state solutions. One can divide all entries of the generator matrix by the negated diagonal entries of the corresponding row, cf. (2). This results in all diagonal entries equalling -1, which can easily be accounted for in the algorithms.
4. As the diagonal entries equal the negative row-sums it is possible to recompute them during the solution process. As the employed algorithms work on the transpose of the generator

Table 2 Memory requirements for FMS(k) generator matrices

	S_i		CSR_i		mCSR_i	
k	div. diag	expl. diag	div. diag	expl. diag	div. diag	expl. diag
5	10.6 MB	10.7 MB	6.9 MB	7.0 MB	6.5 MB	6.6 MB
6	40.1 MB	40.2 MB	26.1 MB	26.2 MB	24.6 MB	24.7 MB
7	129.2 MB	128.8 MB	83.8 MB	83.4 MB	79.1 MB	78.7 MB
8	367.5 MB	364.8 MB	237.5 MB	234.8 MB	224.7 MB	222.0 MB
9	944.9 MB	934.7 MB	609.1 MB	599.0 MB	577.5 MB	567.3 MB
10	2.2 GB	2.1 GB	1.4 GB	1.4 GB	1.3 GB	1.3 GB
11	4.8 GB	4.7 GB	3.1 GB	3.0 GB	2.9 GB	2.9 GB
12	10.0 GB	9.9 GB	6.4 GB	6.3 GB	6.1 GB	6.0 GB
13	24.3 GB	23.5 GB	15.4 GB	14.6 GB	14.8 GB	14.0 GB
14	46.4 GB	44.7 GB	29.3 GB	27.7 GB	28.2 GB	26.6 GB
15	85.1 GB	82.0 GB	53.7 GB	50.6 GB	51.7 GB	48.6 GB

matrix this requires to do the computations column-wise. In addition, this approach would require additional communication in a parallel version. Hence, this technique is not considered any further.

The first three options are available in the PARSECS tool and may be combined with any storage format. For our experiments we used the first approach for transient solutions and the third method for steady-state solutions as defaults.

All sparse storage formats and their variants may be used to store generator matrices in main memory or on disk. The storage of matrices on disk is necessary to solve problems where even the storage format requiring the least memory exceeds the available amount of main memory. Table 2 shows the required memory to store the generator matrix for FMS(k). The storage requirements using the S_i-format are given in column two and three for the variants where one divides the generator matrix by the diagonal (flagged as div. diag) and for the explicit representation of the diagonal using `doubles` (labelled expl. diag). The following columns show the corresponding values for the CSR_i-format (columns four and five) and the mCSR_i-format (columns six and seven). FMS ($k = 13$) was the largest model we were able to generate sequentially using a workstation equipped with 2 GB of main memory; we see that the memory requirements of the mCSR_i-format exceed the available main memory by a factor of seven and, hence, the generator had to be stored on disk.

3.2. Sequential sparse matrix operations

3.2.1. Transposing out-of-core matrices

As seen in Section 2.2, the computation of the steady-state probabilities using linear equation solvers involves computations requiring the transposed generator matrix Q^T. During state-space generation the rate matrix is written to disk using the (indexed) sparse format (S/S_i). As this format requires no special ordering of the entries, transposing a matrix stored using this format is trivial as we only have to exchange the column and the row number of each entry in the corresponding file.

Transposing a rate matrix that fits into main memory is only a sorting problem. Assuming we can read the matrix into an array storing the triples (*col*, *row*, *val*), and sort the entire

array using a sorting algorithm like, for instance, Quicksort [13]. After this step it is a trivial task to transform this array into one of the sparse storage formats.

If we have to transpose a matrix that does not fit into main memory we do this in two steps. At first we do a bucket sort [13], again using the column number as the sort key, that splits the input files into parts that fit into main memory afterwards. In the second step we sort each bucket in main memory and transform it to the required sparse storage format afterwards as for the case where the rate matrix fits into main memory.

As a last step the (separately stored) diagonal entries must be sorted too, as these are written to disk in the order new states are explored. Again Quicksort can be used for this task.

3.2.2. Sequential sparse matrix vector products

The most time consuming operation involved in both the transient and the steady-state solution of CTMCs are matrix vector products (MVPs). Whereas a general MVP of an $n \times n$ matrix and a vector of dimension n is of order $\mathcal{O}(n^2)$, MVPs for sparse matrices can be done in $\mathcal{O}(a)$, assuming a is the number of nonzero entries of the sparse matrix. For generator matrices we typically have $a \ll n^2$.

Below we present algorithms suitable to compute the MVPs for the sparse matrix storage formats presented in Section 3.1. We assume that one wants to compute the MVP $x' \leftarrow Ax$ where $A = R^T$ is the $n \times n$ transposed rate matrix with a off-diagonal nonzero entries. We present the algorithms only for the case where the matrix fits in main memory using the arrays introduced in Section 3.1. Disk-based variants of these algorithms have to use read buffers for these arrays. The algorithms do not change their basic concept but get more complicated due to required index calculations. As done in Section 3.1 we will consider the diagonal later.

Algorithm 1 shows the required steps to calculate an MVP using the S-format. After initialising x' to zero (line 1), it iterates over all nonzero entries (lines 2–5). The variables *cur_row* and *cur_col* (line 3) are used to store the row and the column numbers of the currently treated nonzero entry of the matrix. In line 4 the actual computation is done.

Algorithm 1. (Sparse MVP: S-format)

```
1. x' ← 0; /* 0 is a vector of zeroes */
2. for i = 0, ..., a − 1 do
3.     cur_row ← row[i]; cur_col ← col[i];
4.     x'[cur_row] ← x'[cur_row] + val[i] · x[cur_col];
5. end for;
```

A suitable algorithm for matrices stored in the CSR-format is given below as Algorithm 2. It consists of a for-loop (lines 1–8) iterating over all rows. In contrast to the MVP for the S-format there is no need to initialise the result to zero as we use a temporary *sum* (line 2) to calculate the values of $x'[cur\_row]$ (line 7). The inner for-loop (lines 3–6) treats all nonzero entries of row *cur_row*, which are stored at positions $row\_i[cur\_row], \ldots, row\_i[cur\_row + 1] - 1$ of the arrays *col* and *val*, hence the variable i ranges again from $0 = row\_i[0]$ to $a - 1 = row\_i[n]$. Notice that we use the variant of the CSR-format that stores $n + 1$ row indices in the array *row_i*. The characteristic sum-multiply operation is done in line 5.

Algorithm 2. (Sparse MVP: CSR-format)

```
1. for cur_row = 0, ..., n − 1 do
2.     sum ← 0;
```

```
3.    for i = row_i[cur_row], ..., row_i[cur_row + 1] − 1 do
4.       cur_col ← col[i];
5.       sum ← sum + val[i] · x[cur_col];
6.    end for;
7.    x′[cur_row] ← sum;
8. end for;
```

Using the mCSR-format the MVP algorithm gets slightly more complicated, but resembles the one for the CSR-format. The mCSR-format MVP is shown below as Algorithm 3. It starts with line 0 to keep the line numbering consistent with Algorithm 2. The only difference is the variable *count* (initialised to zero in line 0) that is used to sum the entries of the array *row_n* (second statement in line 7) which replaces the index array *row_i*. In the inner for-loop (lines 3–6) *count* takes the value of *row_i* [*cur_row*] in Algorithm 2. The remaining steps of Algorithm 3 correspond to those in Algorithm 2.

Algorithm 3. (Sparse MVP: mCSR-format)

```
0. count ← 0;
1. for cur_row = 0, ..., n − 1 do
2.    sum ← 0;
3.    for i = count, ..., count + row_n[cur_row] do
4.       cur_col ← col[i];
5.       sum ← sum + val[i] · x[cur_col];
6.    end for;
7.    x′[cur_row] ← sum; count ← count + row_n[cur_row]
8. end for;
```

Until now we did not handle the diagonal entries in the MVPs. We assume the case where the diagonal is stored as a vector d. If we divide the matrix by its diagonal entries all elements of this vector equal -1 and the vector is not stored but used implicitly in the algorithms. For the S-format we can just change the initialisation of x' (line 1 in Algorithm 1) to $x' \leftarrow D \cdot x$, i.e., x_i is multiplied by d_i. For the CSR and mCSR format (Algorithms 2 and 3) we can simply add $D \cdot x$ to the computed x' after these algorithms have finished.

4. Distributed matrix-vector product

Where the previous section focused on the data structure and the sequential sparse matrix vector product algorithm, we now focus our attention on the distributed algorithm. We first summarize the required post-processing steps for the matrices resulting from the distributed state-space generation in Section 4.1, before we present the distributed matrix-vector product algorithm in Section 4.2.

4.1. Generator matrix post-processing

After the state-space generation, the generator matrix Q is distributed in a row-wise fashion among the processors. We associate a sub-matrix $A_{i,j}$ with a corresponding sub-matrix $Q_{i,j}$ of the generator matrix as shown in Fig. 1. These matrices require some post-processing. First, they typically have to be transposed as most equation solvers require Q^T instead of Q (see Section 2.2). Second, they are typically stored in coordinate format as triples

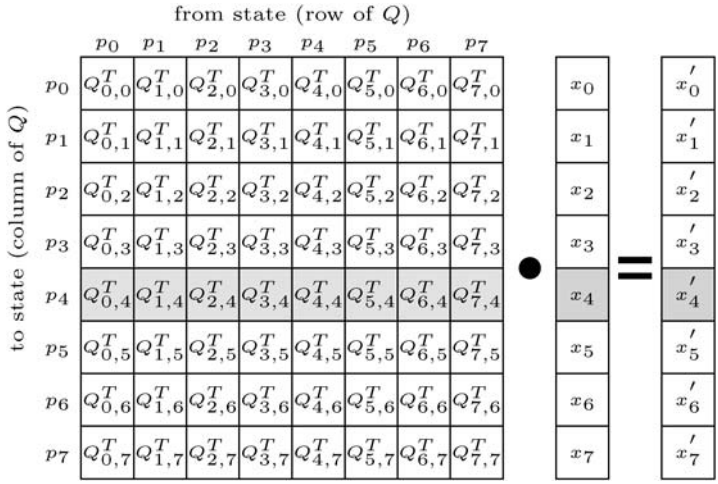

Fig. 1 Distributed matrix-vector product

(*row*, *col*, *value*) and have to be converted to one of the sparse matrix formats introduced in Section 3.1. In the PARSECS tool we use different sparse formats depending on the sparsity of the sub-matrices $Q_{i,j}$. First of all, the matrices $Q^T_{i,i}$, which store the non-cross arcs and, hopefully, contain a large fraction of the overall number of non-zero entries, are stored in the modified compressed sparse row (mCSR) format. Note that cross arcs represent transitions between states i and j that have been allocated to different processors.

For $i \neq j$ the matrices $Q^T_{j,i}$ are reordered to modified compressed sparse column (mCSC) format. The advantage of this format for a parallel MVP, compared to the CSR-format, lies in the fact that entries in the same column (which must be multiplied with the same values of x_j) are stored subsequently. In doing so, we reduce the required communication as these values of x_j must be fetched only once (see Section 4.2). As an exception for sub-matrices $Q^T_{j,i}$ with very few entries (in the extreme case none) are stored in coordinate format. We use this format if it requires less storage than the mCSC format (i.e., whenever $10 \cdot a < 6 \cdot a + n$ (assuming short indices, see Section 3.1). All sparse storage formats are typically used in their indexed variants.

We note that some researchers propose to reorder and remap the states in order to distribute the nonzero entries of the matrix evenly among the processors [24]. As the partitioning functions we used did yield good spatial balance for the analysed models, we did not pursue this further.

4.2. Distributed matrix-vector product

In this section we illustrate the distributed MVP suitable for clusters of workstations as implemented in the PARSECS tool. We assume an MVP of the form $x' \leftarrow Q^T x$, i.e., we compute $Q^T x$, where Q^T is the transpose of the generator matrix of a CTMC and x is a vector of the corresponding dimension, and store the result in the vector x'. As we transposed the generator matrix each processor now stores a row of sub-matrices. The distribution of the sub-matrices and vectors is shown in Fig. 1 for the case of 8 involved processors, where the grey shaded parts are stored by processor 4. Notice that the vectors x and x' are also stored in a distributed way over the processors as indicated.

If we consider the equation $x' \leftarrow Q^T x$ for a certain processor i, it can be split in the following way:

$$x'_i = \underbrace{Q^T_{i,i} x_i}_{\text{task 1 in Algorithm 4}} + \underbrace{\sum_{j \neq i} Q^T_{j,i} x_j}_{\text{task 2 in Algorithm 4}} \qquad (6)$$

This leads to the distinction between two cases during the computation of the MVP, "local computations", i.e., the first addend in (6), and computations requiring communication with other processes, called "remote computations", i.e., the second addend in (6). Remember that the matrices $Q^T_{i,i}$ contain the nonzero entries which do not correspond to cross-arcs. Hence, typically 50% or more of all the nonzero entries are located in the diagonal submatrices and for this reason more than half of the required floating point operations require no communication.

The framework of a distributed MVP algorithm is given as Algorithm 4. After the initialisation of the result to zero (line 1) the local (task 1) and remote (task 2) computations are done in parallel. This either means that they are realized as separate threads in which two copies of x'_i are required which are added after both threads have finished, or that these computations are interleaved in some way in order to overlap computation and communication. In fact, the latter method has been implemented in the PARSECS tool as the underlying MPI implementation (MPICH) is not thread-safe. The local computations are implemented as standard sequential (out-of-core) sparse MVPs as described in Section 3.2.2 using the indexed mCSR-format for the storage of $Q^T_{i,i}$.

The remote computations are handled in task 2 (lines 5–8), where the sub-MVPs $Q^T_{j,i} x_j$ are computed. This corresponds to the second addend of Eq. (6) and a single sub-MVP is given by the term $Q^T_{j,i} x_j$ (line 7). Details of such a remote sub-MVP will be given later as Algorithm 5. Note that the for-loop starts with $j \leftarrow i + 1$ (mod `noprocs`) in order to balance the communication requirements between all processes by assuring that not all processors start communicating with the same pattern at the same time. For a single sub-MVP $Q^T_{j,i} x_j$, process i requires the entries from x_j that correspond to non-empty columns of the matrix $Q^T_{j,i}$.

Algorithm 4. (Distributed MVP for processor *i*)

```
1.  x'_i ← 0;
2.  do parallel
3.     task 1: local comp. */
4.        x'_i ← x'_i + Q^T_{i,i} x_i;
5.     task 2: \* remote comp. */
6.        for j ← i + 1, ..., i + noprocs − 1 (mod noprocs) do
7.           x'_i ← x'_i + Q^T_{j,i} x_j; /* see Algorithm 5 */
8.        end for;
9.  end do parallel;
```

We assume that the matrices $Q^T_{j,i}$ are very sparse and therefore it is profitable to request only the required values of the vector x_j from processor j instead of all successive elements. Experiments using fast Ethernet as communication platform support this assumption. Hence, a matrix format in which the entries are sorted column by column seems most appropriate. If we used a format which stores the entries sorted by rows we would have to request certain

entries of the (remotely stored) vector x_j multiple times. At the same time, we want to keep the memory requirements for a sub-matrix $Q_{j,i}^T$ as low as possible. In contrast to complete generator matrices, as considered in the sequential case, these matrices may contain empty rows and columns, hence we consider the coordinate format (sorted by columns) in addition to the modified compressed sparse column format. When using these formats with indices of 2 bytes, the sub-matrix $Q_{j,i}^T$ of dimension $n_j \times n_i$ with m non-zero entries can either be stored in coordinate format using $10 \cdot m$ bytes or in mCSC format using $6 \cdot m + n_i$ bytes. We use the format that requires less memory.

Algorithm 5 shows the pseudo-code for the computation of a certain sub-MVP $Q_{j,i}^T x_j$ on processor i. In order to ease notation we will only consider the remote sub-MVP for matrices stored in modified CSC format as the required modifications for the coordinate format or indexed storage formats are obvious. We assume, that a function *read_cols*(`no_cols`) exists which reads a portion of $Q_{j,i}^T$ from disk (`no_cols` column indices and the corresponding row numbers and values; this is a constant that allows us to tune the memory usage and the performance of the MVP). This function returns the number of actually read columns (this might be less than `no_cols` for the last read operation) and fills the arrays `col_entries`, `row` and `val`. In addition to these variables we use an array of integers `req` for the positions of the required entries of the (remotely stored) vector x_j and an array of doubles `x_rem` for the values stored at those positions; both arrays are of dimension `no_cols`. The variables `cur_col` and `cur_row` store the column, respectively, the row number we currently work on and `cur_rowi` holds the current index to the `row` array.

Algorithm 5. (remote sub-MVP for $x_i' \leftarrow x_i' + Q_{j,i}^T x_j$).

```
1.  cur_col ⟵ 0;
2.  while not(EOF(Q^T_{j,i})) do
3.    rc ⟵ read_cols(no_cols);
4.    no_req ⟵ 0;
5.    for c ← 0, ..., rc − 1 do
6.      if col_entries[c] ≠ 0
7.        req[no_req] ⟵ c + cur_col;
8.        no_req ⟵ no_req + 1;
9.      end if;
10.   end for;
11.   x_rem ⟵ Request(j, req);
12.   no_req ⟵ 0; cur_rowi ⟵ 0;
13.   for c ← 0, ..., rc − 1 do
14.     if col_entries[c] ≠ 0
15.       for d ⟵ 0, ..., col_entries[c] − 1 do
16.         cur_row ⟵ row[cur_rowi];
17.         x_i[cur_row] ⟵ x_i[cur_row]
                          +x_rem[no_req] · val[cur_rowi];
18.         cur_rowi ⟵ cur_rowi + 1;
19.       end for;
20.       no_req ⟵ no_req + 1;
21.     end if;
21.   end for;
22.   cur_col ⟵ cur_col + rc;
23. end while;
```

The algorithm consists of a while loop (lines 2–24) which assures that the complete matrix $Q^T_{j,i}$ is read. During each iteration at most `no_cols` columns are read in line 3. The actually read number of columns is returned and stored in the variable *rc*. After setting the number of required remote entries (from the vector x_j) to zero (line 4) the for loop (lines 5–10) iterates over the read columns and fills the array `req` with the column numbers which are non-empty (see line 6; note that empty columns may appear as we are operating on sub-matrices of a CTMC generator) and, hence, correspond to the positions of the vector x_j required. The actual communication is done in line 11 where we request the required entries of x_j from process j and store them in `x_rem`. For each occurring column number, 12 bytes have to be transferred, since the column number is sent as an integer (4 byte) and the value is received as a double (8 byte). It is important to guarantee that this communication is overlapped with some computation in order to achieve good performance. This can either be accomplished by doing local computations while a process "waits" for requested values or by implementing some kind of windowing protocol, that allows to request several chunks of data before the requests have been answered. In the PARSECS tool, the first approach has been chosen. Using non-blocking send and receive operations allows us to perform local computations during that communication phase. The request for certain elements of a remotely stored vector in line 11 is implemented by a non-blocking send operation that sends the indices of the required vector elements, directly followed by a corresponding non-blocking receive function call that receives the appropriate elements of the remotely stored vector while program execution continues. Afterwards, we transfer the program execution to the local computations (task 1 in Algorithm 4) where we check at regular intervals whether the vector elements were received. In line 12 we reset the variables `no_req` and `cur_rowi` to zero in order to use them as indices to the received array `x_rem` and the arrays `row` and `val`, respectively. The for loop spanning lines 13–21 does the actual computation. First (line 14) we check whether the current column is non-empty. If this is true, we loop over the number of rows corresponding to that column (lines 15–19). In this loop we compute the actual row number `cur_row` in line 16, do the multiply-add operation (line 17) and increment the index to the `row` and `val` arrays in line 18. In line 20 the index to the received entries of the x_j vector, which will be handled in the next iteration of the for loop (comprising lines 15–19), is incremented. After all data read by line 3 has been used, we add the number of handled columns to `cur_col` (line 22). Additional implementation details for this algorithm can be found in [5].

5. Experimental results

In this section we present experimental results obtained using a cluster of workstations (cf. Section 5.1). Some background on how to evaluate parallel and distributed programs is given in Section 5.2. Section 5.3 comprises results for the FMS model. More results can be found in [5].

5.1. Computing environment

All our experiments were performed on the cluster of the department for computer science at the RWTH Aachen comprising a server and 26 clients (nodes). Each node is a dual 500 MHz Pentium III equipped with 512 MB main memory and 40 GB hard disk. The nodes are connected via fast ethernet. Whenever we do not use all 52 processors, we use at most one processor per node.

5.2. Evaluation of parallel and distributed programs

Beside absolute execution times we measure the quality of our programs in terms of **speedup** and **efficiency** [17], which we define below.

Let $T(1)$ be the execution time of a one-processor solution and $T(N)$ for $N \geq 2$ the execution time of the parallelised program on an N processor system. Then, the fraction $S(N) = T(1)/T(N)$ is called the achieved **speedup**. Ideally the work is equally distributed and no additional overhead occurs, resulting in a speedup $S(N) = N$. This is called **linear speedup**; it is difficult to obtain in practice. The fraction $E(N) = S(N)/N$ is called the achieved **efficiency** of the parallelised program. When comparing the runtime of a parallel program with a specialised sequential solution, in contrast to a parallel solution running on only one processor, one speaks of **absolute** speedup and efficiency; when referring to these, we ran the sequential program on a single workstation of the cluster.

5.3. FMS model results

In what follows we present results for the steady-state analysis of the FMS model. The execution time of the solvers largely depends on the performance of the MVPs. In the distributed setting the performance of the MVPs depends strongly on the partitioning function. Hence, we first discuss the influence of the partitioning function on single MVPs in Section 5.3.1, before we address results for the actual computation of the steady-state distributions in Section 5.3.2.

5.3.1. MVP performance in relation to state-space partitioning

When using 100 Mbit Ethernet the overall performance of the MVP depends highly on the required communication which in turn depends on the number of cross arcs (or cross-transitions). These are transitions between states i and j that have been allocated to different processors, so that in the distributed computation, processor i needs probability information from processor j.

To illustrate this, we experimented with different partitioning functions for the same model. In Table 3 we present the results for FMS ($k = 11$) and eight processors for three different partitioning function which generated 15.3%, 37.3% and 72.4% cross arcs, respectively. Columns 2–3 list the state and arc imbalance factors, defined as the quotient of the size of the largest partition of states or arcs and the average partition size. Column four of Table 3 shows the required time per MVP dependent of the fraction of cross arcs, which is given in the first column. Columns 5–6 list the bandwidth of the network consumed by MVPs, i.e., the total number of bytes per second sent and received by all participating workstations, as well as averaged per node. The numbers given do not contain any overhead introduced by MPI or TCP/IP. The corresponding values concerning required disk I/O are given in columns 7–8. Note that the actual maximum values for a single node for the "best" partitioning function (line one) were 28% larger than the given average for the required network bandwidth and 15% larger for the disk reads due to the imbalance of states and arcs. From these results it becomes clear that, at least on workstation clusters, it is important to keep the fraction of cross arcs low, even at the cost of higher imbalance factors.

Table 3 MVP speed in dependence of cross-arcs, FMS ($k = 11$, *noprocs* $= 8$).

	imbalance			bandw. (MByte/s)		disk (MByte/s)	
cross-arcs	states	arcs	time per MVP	aggreg.	node	aggreg.	node
15.3%	1.0256	1.0605	56.1s	35.553	4.444	62.61	7.82
37.3%	1.0668	1.0602	74.6s	55.073	6.884	54.51	6.81
72.4%	1.0006	1.0004	104.0s	61.937	7.742	42.83	5.35

5.3.2. Distributed steady-state solution

Using the MVP described in Section 4, the algorithms for steady-state solution can be implemented in a straightforward fashion. In the following we just give some results to elucidate which models are *solvable* and how long this takes.

First of all, we were able to compute the steady-state solution for all models for which we generated the state-spaces. The largest CTMC we were able to solve is FMS($k = 15$) which has 724 million states. The steady-state solution required slightly over 12 days, achieving an accuracy (residual criterion) of 10^{-8}. During this time 231.4TB were read from the disks, yielding an aggregate throughput of 172 MB/s (6.6 MB/s per node) and 167.2 TB were sent and received over the net. This means each node sent and received about 4.7 MB/s.

To discuss speedup we have to restrict the analysed models to cases that can be handled sequentially on a single node of the cluster. As these are only equipped with 512 MB of memory we present the attained speedups for the method of Jacobi for FMS($k = 10$) and for the CGS method for FMS($k = 9$) in Fig. 2. Note that for the latter case the sequential version had to use a specialised implementation of CGS which saves intermediate vectors to disk. This explains the much better speedups for this case, compared to the Jacobi solution for the case $k = 10$. In general one observes that the benefit of the use of more than one processor per node, i.e., more than 26 processors, is small. We attribute this result to the fact that when using two processors in a single node these processors have to share the memory, hard disk and especially network bandwidth. The benefit of the tight connection between two processors in a single node is negligible when using more than two nodes. Note that these speedups were achieved using a partitioning function comparable to the middle one from Table 3. Using a better partitioning function, slightly better speedups are achievable.

Table 4 gives an overview of the solution times for FMS(k). All tests were done using 26 processors. Columns 2–5 show the statistics for the method of Jacobi where we computed the

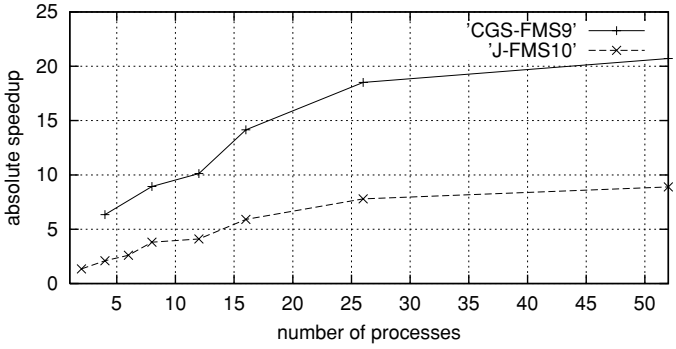

Fig. 2 Absolute speedups for the solution of the FMS model

Table 4 Performance results for the distributed steady-state solution of the FMS model (time in h:mm:ss) using 26 processors

	Jacobi				CGS			
	$\epsilon = 10^{-6}$		$\epsilon = 10^{-9}$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-15}$	
k	steps	time	steps	time	steps	time	steps	time
6	886	0:06:24	1507	0:10:53	371	0:05:22	893	0:12:55
7	1034	0:21:06	1783	0:36:23	150	0:07:09	1253	0:51:28
8	1181	0:43:56	1977	1:13:34	503	0:40:19	1469	1:57:44
9	1336	2:31:54	2208	4:11:03	863	2:52:59	1623	5:25:20
10	1492	7:36:36	2466	12:34:41	903	9:25:31	3559	37:08:53
11	1652	18:21:37	2722	30:15:08	665	15:04:44	3124	70:50:13
12	1818	47:59:51	2979	78:38:59	1219	65:00:17	2184	116:27:52
13	1990	68:53:55	3257	112:45:56	857	90:58:09	1987	210:55:02
14	2169	141:08:33	4448	289:26:34	325	72:36:53	–	–
15	2335	292:16:37	–	–	–	–	–	–

solution up to accuracies (measured as the relative residual criterion (maximum norm of the residual divided by the maximum norm of the solution vector)) of $\epsilon = 10^{-6}$ and $\epsilon = 10^{-9}$. For both accuracies we give the number of iteration steps performed (columns two and four) and the required wall-clock time (columns three and five). We were not able to achieve a better accuracy than about $\epsilon = 5 \cdot 10^{-10}$ using the method of Jacobi (this was also the case for the sequential solution). For the CGS method we used accuracies of $\epsilon = 10^{-6}$ (columns 6–7), to present result comparable to the method of Jacobi and $\epsilon = 10^{-15}$ (columns 8–9) which lies in the range of the machine precision. Again, for both accuracies we present the number of required iteration steps and the wall-clock time required to perform these. Note that we used a special version of the CGS solver that uses only three vectors in memory, while the other vectors are written to disk, to compute the results for $k = 14$.

If we compare the results with the ones from the sequential version, we observe that the required number of CGS steps is much more irregular than in the sequential version. Apparently, one can be extremely lucky, like in the cases $k = 7$ and $k = 14$, where the accuracy $\epsilon = 10^{-6}$ was achieved very quickly, but on the other hand sometimes the required number of iterations is extremely high, like in the case $k = 10$ for the accuracy $\epsilon = 10^{-15}$. We note that the number of CGS iteration steps depends on the initial guess of the solution, which we take as a random vector that we normalise to 1. Hence, the actually required number of iterations varies between different runs.

6. Conclusion

In this paper we presented sparse data structures and distributed algorithms for the efficient disk-based solution of very large Markov chains on a cluster of workstations, using the MPICH message passing library. We have reported extensive measurements on a 26 dual-node cluster, and show that CTMCs with up to 724 million states can be solved.

We show that all the necessary computations (both for performance and dependability evaluation as well as for CSL model checking) can be described as series or sums of matrix-vector multiplications, and that these can be executed efficiently in a distributed way on a cluster of workstations. Thus, we can analyse larger models than ever before, and we can do so faster. We also show that good speedups are obtained, however, not as good as for distributed state-space generation and distribution CTL model checking. Although

the presented algorithms increase the size of models that can be analysed significantly, the absolute times necessary to solve the largest models still appear too high for widespread practical application.

Future work includes the analysis of our algorithms on other types of computing platforms and machines, the integration with our distributed algorithms for CTL model checking [7] (leading to distributed algorithms for CSL and possibly CSRL model checking), and the incorporation of symbolic representation schemes for the matrix Q.

Appendix

In this appendix we present results for the distributed evaluation of another CTMCs originating from a GSPN, namely the kanban model.

A.1. Results for the kanban model

The kanban model was introduced by Ciardo and Tilgner in [11] in two variants, one with timed and one with immediate synchronisation. We only discuss the variant with immediate synchronisation here. It is parameterised with the initial number of tokens N.

For the distributed computation of the steady-state distribution we present absolute speedups for the cases $N = 6$ (4 785 536 states) and $N = 7$ (15 198 912 states) in Fig. 3. For the smaller case the achieved speedups are unsatisfactory, as 16 processors are required to obtain an absolute speedup of slightly less than two and even using 26 processors the speedup does not exceed 2.3. But we note that we can solve this model (until convergence stagnates) in less than an hour on a cluster node and in about 35 min on a workstation using a 1 GHz Pentium III processor. For $N = 7$ we realise an absolute speedup corresponding to an efficiency of at least 50% if we use less than 8 processors. For 16 and 26 processors the speedups still grow and the corresponding efficiencies are roughly 40% and 30%, respectively.

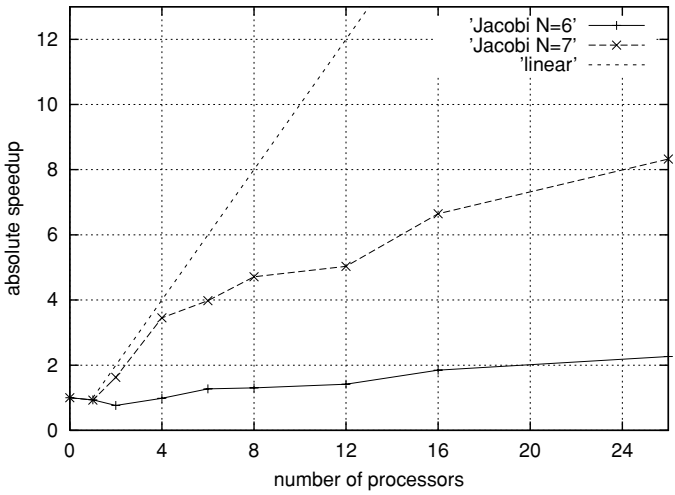

Fig. 3 Absolute speedups for the solution step, kanban model

Table 5 Kanban model, steady-state solution using 16 processors

			Jacobi, $\delta = 10^{-4}$		CGS, $\delta = 10^{-6}$		CGS, $\delta = 10^{-10}$	
N	n	a	Time	Steps	Time	Steps	Time	Steps
8	$42{\cdot}10^6$	$457{\cdot}10^6$	6:37:24	728	1:17:01	142	10:42:13	832
9	$106{\cdot}10^6$	$1177{\cdot}10^6$	12:45:46	789	09:25:05	167	53:41:21	952
10	$244{\cdot}10^6$	$2770{\cdot}10^6$	73:25:58	853	44:25:33	187	271:32:38	1143

For the cases we did not address sequentially, we computed the steady-state distribution of the kanban model for $N = 8,\ldots,10$ using 16 processors. Although a solution for $N = 11$ and $N = 12$ would have been possible, we decided not to compute those steady-state distributions as the required computation times are enormous. Table 5 presents the required wall-clock times in format hours:min:sec and number of iterations required for the solution process for a given parameter N shown in column one. Column two and three show the resulting number of states n and non-zeroes a respectively. The next two columns show the results for the method of Jacobi, where we stopped the computation after an accuracy of $\delta = 10^{-4}$ was reached. For the case $N = 8$ we computed more Jacobi iterations but observed stagnating convergence after 1800 iterations, where an accuracy of $1.2{\cdot}10^{-5}$ was reached. Columns four to seven show the required time and steps when using the CGS method and stopping after reaching an residual norm of $\delta = 10^{-6}$ and $\delta = 10^{-10}$ respectively.

For these larger cases we also studied the impact of using 26 instead of 16 processors, which offered a performance increase of roughly 30–40% (where 62.5% is the theoretical maximum), hence, we conclude that for these larger model sizes the solution process scales quite well.

Acknowledgments The work presented in this paper has largely been performed in the period 1999–2003 when the authors were working in the Department of Computer Science of the RWTH Aachen, Germany. The work was supported by the DFG (projects HA2666/1-1,2).

References

1. Ajmone Marsan M, Conte G, Balbo G (1984) A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Trans Comp Syst 2(2):93–122
2. Ajmone Marsanan M, Balbo G, Conte G, Donatelli S, Franceschinis G (1995) Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing. Wiley
3. Allmaier S, Kowarschik M, Horton G (1997) State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In: Proceedings of the 7th International Workshop on Petri Nets and Performance Models. pp 112–121
4. Baier C, Haverkort B, Hermanns H, Katoen J (2003) Model-checking algorithms for continuous-Time Markov Chains. IEEE Trans Softw Eng 29(6):524–541
5. Bell A (2004) Distributed evaluation of stochastic Petri nets. Ph.D. thesis, Rheinisch-Westfalische Technischen Hochschule Aachen. ISBN-Nr.: 3-930376-36-9
6. Bell A, Haverkort B (2001) Serial and parallel out-of-core solution of linear systems arising from generalised stochastic Petri net models. In: Tentner A (ed) Proceedings high performance computing symposium HPC 2001. pp 242–247
7. Bell A, Haverkort B (2005) Sequential and distributed model checking of Petri nets. Softw Tools Technol Transf 7(1):43–60
8. Carney S, Heroux M, Li G (1993) A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90 (Revised), CERFACS, Toulouse, France
9. Ciardo G, Fricks R, Muppala J, Trivedi K (1994) SPNP Reference Guide Version 3.1
10. Ciardo G, Gluckman J, Nicol D (1998) Distributed state space generation of discrete-state stochastic models. INFORMS J Comput 10(1):82–93

11. Ciardo G, Tilgner M (1996) On the use of Kronecker Operators for the Solution of Generalized Stochastic Petri Nets, ICASE, No. 96-35
12. Ciardo G, Trivedi K (1993) A decomposition approach for stochastic reward Net Models. Perform Eval 18(3):37–59
13. Cormen T, Leiserson C, Rivest R (1990) Introduction to Algorithms. MIT Press
14. Deavours D, Sanders W (1998) An efficient disk-based tool for solving large Markov Models. Perform Eval 33(1):67–84
15. Dutt I, Grimes R, Lewis J (1992) 'Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical Report RAL 92-086, Rutherford Appleton Laboratory, Chilton, Oxon, England
16. Forum MPI (1994) MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee Computer Science Department
17. Foster I (1995) Designing and building parallel programs. Addison-Wesley
18. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput 22(6):789–828
19. Gropp WD, Lusk E (1996) User's Guide for `mpich` , a portable implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory. ANL-96/6
20. Haverkort B (1998) Performance of computer communication systems: A model-based approach. Wiley
21. Haverkort B, Bell A, Bohnenkamp H (1999) On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri Nets. In: Proceedings of the 8th International Workshop on Petri Nets and Performance Models. pp 12–21
22. Hermanns H, Kwiatkowska M, Norman G, Parker D, Siegle M (2003) On the use of MTBDDs for performability analysis and verication of stochastic systems. J Log Algeb Progr: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems 56(1–2):23–67
23. Hermanns H, Meyer-Kayser J, Siegle M (1999) Multi-terminal binary decision diagrams to represent and analyse continuous time markov chains. In: Plateau B, Stewart W, Silva M (eds) Proceedings of the 3rd Int. workshop on the numerical solution of Markov chains. pp 188–207
24. Knottenbelt W, Harrison P (1999) Distributed disk-based solution techniques for large Markov models. In: Proceedings of the 3rd International meeting on the numerical solution of Markov chains. pp 58–75
25. Knottenbelt W, Harrison P, Mestern M, Kritzinger P (2000) A probabilistic dynamic technique for the distributed generation of very large state spaces. Perform Eval 39(1–4):127–148
26. Knottenbelt W, Mestern M, Harrison P, Kritzinger P (1998) Probability, Parallelism and the State Space Exploration Problem. In: Puigjaner R, Savino N, Serra B (eds) Computer performance evaluation, Vol 1469 of Lecture Notes in Computer Science. pp 165–179
27. Kwiatkowska M, Mehmood R (2002) Out-of-core solution of large linear systems of equations arising from stochastic modelling. In: Proc PAPM-PROBMIV, Vol. 2399 of Lecture Notes in Computer Science. pp 135–151
28. Kwiatkowska M, Mehmood R, Norman G, Parker D (2002) A symbolic out-of-core solution method for markov models. Electr Notes Theor Comput Sci 68(4)
29. Kwiatkowska M, Parker D, Zhang Y, Mehmood R (2004) Dual-processor parallelisation of symbolic probabilistic model checking. In: Proceedings IEEE MASCOTS. pp 123–130
30. Marenzoni P, Caselli S, Conte G (1997) Analysis of large GSPN models: a distributed solution tool. In: Proceedings of the 7th International workshop on petri nets and performance models. pp 122–131
31. Stewart W (1994) Introduction to the mumerical solution of Markov chains. Princeton University Press