(accepted for publication on INFORMS Journal on Computing)

Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models*

Peter Buchholz
Department of Computer Science
Dresden University of Technology
D-01062 Dresden, Germany
p.buchholz@inf.tu-dresden.de

Gianfranco Ciardo
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
ciardo@cs.wm.edu

Susanna Donatelli
Dipartimento di Informatica
Università di Torino
Corso Svizzera 185, 10149 Torino, Italy
susi@di.unito.it

Peter Kemper
Informatik IV
Universität Dortmund
D-44221 Dortmund, Germany
kemper@ls4.informatik.uni-dortmund.de

Abstract

We present new algorithms for the solution of large structured Markov models whose infinitesimal generator can be expressed as a Kronecker expression of sparse matrices. We then compare them with the shuffle-based method commonly used in this context and show how our new algorithms can be advantageous in dealing with very sparse matrices and in supporting both Jacobi-style and Gauss-Seidel-style methods with appropriate multiplication algorithms. Our main contribution is to show how solution algorithms based on Kronecker expression can be modified to consider probability vectors of size equal to the "actual" state space instead of the "potential" state space, thus providing space and time savings. The complexity of our algorithms is compared under different sparsity assumptions. A nontrivial example is studied to illustrate the complexity of the implemented algorithms.

1 Introduction

Continuous time Markov chains (CTMCs) are an established technique to analyze the performance, reliability, or performability of dynamic systems from a wide range of application areas. CTMCs

*G. Ciardo's research was partially supported by the National Aeronautics and Space Administration under NASA Contracts No. NAS1-19480 and NAG-1-2168. P. Kemper's research was partially supported by Deutsche Forschungsgemeinschaft, SFB 559. In addition, P. Buchholz, S. Donatelli, and P. Kemper received travel support from DAAD Germany and CRUI Italy under the bilateral program Vigoni.

are usually specified in a high-level modeling formalism, then a software tool is employed to generate the state space and generator matrix of the underlying CTMC and compute the stationary probability vector, from which most quantities of interest can be obtained as a weighted sum by using "reward rates" as weights [20].

Although the mapping of a high-level model onto the CTMC and the computation of the stationary distribution are conceptually simple, practical problems arise due to the enormous size of CTMCs modeling realistic systems. Sophisticated generation and analysis algorithms are required in practice.

In this paper, we consider the stationary solution of large ergodic CTMCs, that is, the computation of the vector $\boldsymbol{\pi} \in I\!R^{|\mathcal{T}|}$, where $\boldsymbol{\pi}_i$ is the stationary probability of state i and $\mathcal{T}$ is the set of states of the CTMC, or (actual) state space. $\boldsymbol{\pi}$ is the solution of the system of linear equations

$$\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0} \qquad \text{subject to} \qquad \boldsymbol{\pi} \cdot \mathbf{1}^T = 1, \tag{1}$$

where $\mathbf{Q}$ is the generator matrix of the CTMC. However, our contributions can also be used for other analyses such as the computation of the expected time spent in transient states up to absorption in absorbing CTMCs and the transient analysis of arbitrary CTMCs [9].

Direct solution methods such as the well-known Gaussian elimination are not applicable for the solution of Eq. 1, since their fill-in results in excessive memory requirements. Iterative techniques based on sparse storage schemes for $\mathbf{Q}$ are more appropriate, but even they are memory-bound when applied to realistic examples. Virtual memory is of little help, since access times to virtual memory are too long to allow an efficient implementation of iterative solution techniques (although [14] reports some encouraging results).

Recently, solution techniques for CTMCs have been developed that compute $\boldsymbol{\pi}$ without generating and storing $\mathbf{Q}$ explicitly. The idea is to represent $\mathbf{Q}$ as a submatrix of a matrix $\hat{\mathbf{Q}}$ given as a sum of Kronecker products of smaller matrices resulting from a high-level model structured into submodels. The method has been applied to several high-level formalisms where models are described in a compositional way [16, 17, 21, 25]. Solution methods exploiting a Kronecker structure are iterative but they differ from conventional iterative techniques in how they perform the required vector-matrix multiplications.

A first approach [25] employed the slowly-converging Power method, used dense storage schemes for the submodel matrices, and computed the solution using the "potential" state space, a (possibly much larger) superset of the actually reachable states. Event rates could be either constant or, for enhanced modeling power, they could depend on the state (functional rates). Several extensions and generalizations have been introduced since then. In [19], Fernandes et al. describe how to treat functional rates without adding excessive overhead. [8, 6, 28, 29, 30] present Kronecker adaptations of iterative solution methods more advanced than just the Power method. Matrices involved in the Kronecker approach are typically sparse; the effect of sparsity for the shuffle-based algorithm has been discussed in [19], where the authors indicate that for extremely sparse matrices other multiplication algorithms can be more efficient that the shuffle-based one. We will give a comparison of different sparse matrix algorithms but consider also the impact of unreachable states. The overhead of unreachable states was studied in [12, 21], concluding that

Symbol	Definition or properties	Meaning
M_k		k-th submodel
n_k	$n_k \geq 2$	Number of local states for M_k
n_l^u	$\prod_{k=l}^{u} n_k$	Number of potential states for $M_{[l,u]}$
n	n_1^K	Number of overall potential states
$\bar{n}_k$	$n/n_k = n_1^{k-1} \cdot n_{k+1}^K$	Number of potential states when M_k is ignored
$\hat{\mathcal{T}}^k$, $\mathcal{T}^k$	$\{0, \ldots, n_k - 1\}$	Potential, actual local state space (s. s.) for M_k
$\hat{\mathcal{T}}_1^k$	$\mathcal{T}^1 \times \mathcal{T}^2 \times \cdots \times \mathcal{T}^k$	Potential s. s. for $M_{[1,k]}$
$\hat{\mathcal{T}}, \mathcal{T}$	$\hat{\mathcal{T}} = \hat{\mathcal{T}}_1^K, \mathcal{T} \subseteq \hat{\mathcal{T}}$	Potential and actual overall s. s.
$\mathcal{T}_1^k$	$\{i_{[1,k]} : \exists i_{[k+1,K]}, i_{[1,K]} \in \mathcal{T}\}$	Projection of the actual s. s. $\mathcal{T}$ on $M_{[1,k]}$
$\mathcal{T}^k(i_{[1,k-1]})$	$\{i_k : i_{[1,k]} \in \mathcal{T}_1^k\}$	Actual s. s. for M_k when M_l is in state i_l, $\forall 1 \leq l < k$
Ψ	$\Psi : \hat{\mathcal{T}} \to \{0, \ldots, \lvert\mathcal{T}\rvert - 1, \mathsf{null}\}$	Position of a state in lexicographic order
$\hat{\boldsymbol{\pi}}, \boldsymbol{\pi}$	$\forall i \in \mathcal{T}\ \hat{\boldsymbol{\pi}}_i = \boldsymbol{\pi}_{\Psi(i)}$	Steady state probability vector
$\hat{\mathbf{Q}}, \mathbf{Q}, \hat{\mathbf{R}}, \mathbf{R}$	$\mathbf{Q} = \hat{\mathbf{Q}}_{\mathcal{T},\mathcal{T}}, \mathbf{R} = \hat{\mathbf{R}}_{\mathcal{T},\mathcal{T}}$	Infinitesimal generator, transition rate matrix
$\hat{\mathbf{h}}, \mathbf{h}$	$\forall i \in \mathcal{T}\ \hat{\mathbf{h}}_i = \mathbf{h}_{\Psi(i)} = \hat{\mathbf{Q}}_{i,i}^{-1}$	Expected holding time vector

Table 1: Symbols used in the paper

it can outweight the advantages of a Kronecker structure by far, so that appropriate treatment of unreachable states is essential to achieve algorithms that perform well even for the non-ideal case where the potential state space contains many unreachable states.

We follow this line here and present a family of solution techniques that use sparse storage for the submodel matrices and iteration vectors of the size of the actual state space, and we consider both the Jacobi and the Gauss-Seidel methods. Additionally, we compare the complexity of different Kronecker-based vector-matrix multiplication algorithms, both theoretically and by means of a realistic example. Modifications of the algorithms to manage functional rates are also discussed.

The next section defines our notation and the Kronecker expression considered for the generator matrix, and introduces a running example. Sect. 3 presents and analyzes different algorithms to multiply a vector by a matrix represented as a Kronecker product, using our running example. Sect. 4 discusses modifications of the algorithms of Sect. 3 that consider only the set of reachable states. Sect. 5 compares the various multiplication algorithms and briefly discusses the treatment of functional rates, while Sect. 6 describes iterative solution approaches that use these multiplication algorithms to compute the stationary solution of a CTMC.

2 Notation

Table 1 summarizes the symbols we use. Except for the set of real numbers, $\mathbb{R}$, all sets are denoted by upper-case calligraphic letters (e.g., $\mathcal{A}$); row vectors and matrices are denoted by lower- and

upper-case bold letters, respectively (e.g., $\mathbf{x}$, $\mathbf{A}$); their entries are indexed starting from 0 and are denoted by subscripts (e.g., $\mathbf{x}_i$, $\mathbf{A}_{i,j}$); a set of indices can be used instead of a single index, for example, $\mathbf{A}_{\mathcal{X},\mathcal{Y}}$ denotes the submatrix of $\mathbf{A}$ corresponding to set of rows $\mathcal{X}$ and the set of columns $\mathcal{Y}$. We also denote families of like-quantities with subscripts, if scalars, or superscripts, if sets, vectors, or matrices (e.g., x_i or $\mathbf{x}^i$), and use a shorthand "range" notation to indicate sequences of them (e.g., $x_{[1,n]} = x_1, \ldots, x_n$).

$\eta[\mathbf{A}]$ denotes the number of nonzero entries (nonzeros) in matrix $\mathbf{A}$. $\mathbf{0}_{x\times y}$ and $\mathbf{1}_{x\times y}$ denote matrices with x rows and y columns, having all entries equal 0 or 1, respectively, while $\mathbf{I}_x$ denotes the identity matrix of size $x \times x$; the dimensions of these matrices are omitted if clear from the context. Given a vector $\mathbf{x}$, $diag(\mathbf{x})$ is a square matrix having vector $\mathbf{x}$ on the diagonal and zero elsewhere. Given an $n \times n$ matrix $\mathbf{A}$, $rowsum(\mathbf{A}) = diag(\mathbf{A} \cdot \mathbf{1}_{n\times 1})$ is a matrix having the diagonal equal to the sums of the entries on each row of $\mathbf{A}$, and zero elsewhere.

We recall the definition of the Kronecker product $\mathbf{A} = \bigotimes_{k=1}^{K} \mathbf{A}^k$ of K square matrices $\mathbf{A}^k \in \mathbb{R}^{n_k \times n_k}$. Let $n_l^u = \prod_{k=l}^{u} n_k$, $n = n_1^K$, and $\bar{n}_k = n/n_k$. We use a mixed-base numbering scheme where the tuple $l_{[1,K]}$ corresponds to the number $(\ldots((l_1)n_2 + l_2)n_3 \cdots)n_K + l_K = \sum_{k=1}^{K} l_k n_{k+1}^K$ (letting $n_{K+1}^K = 1$), and vice versa. If $i_{[1,K]}$ and $j_{[1,K]}$ are the mixed-based representation of i and j, respectively, the generic element of $\mathbf{A} \in \mathbb{R}^{n\times n}$ is

$$\mathbf{A}_{i,j} = \mathbf{A}_{i_{[1,K]},j_{[1,K]}} = \mathbf{A}^1_{i_1,j_1} \cdot \mathbf{A}^2_{i_2,j_2} \cdots \mathbf{A}^K_{i_K,j_K}. \tag{2}$$

The Kronecker sum $\bigoplus_{k=1}^{K} \mathbf{A}^k$ is defined in terms of Kronecker products, as

$$\bigoplus_{k=1}^{K} \mathbf{A}^k = \sum_{k=1}^{K} \mathbf{I}_{n_1} \otimes \cdots \otimes \mathbf{I}_{n_{k-1}} \otimes \mathbf{A}^k \otimes \mathbf{I}_{n_{k+1}} \otimes \cdots \otimes \mathbf{I}_{n_K} = \sum_{k=1}^{K} \mathbf{I}_{n_1^{k-1}} \otimes \mathbf{A}^k \otimes \mathbf{I}_{n_{k+1}^K}.$$

We are interested in algorithms that exploit sparsity. For the Kronecker product, the number of nonzeros is $\eta[\bigotimes_{k=1}^{K} \mathbf{A}^k] = \prod_{k=1}^{K} \eta[\mathbf{A}^k]$. For the Kronecker sum, diagonal entries in the matrices $\mathbf{A}^k$ might result in merged entries on the diagonal of $\mathbf{A}$, thus we can only bound the number of nonzeros, $\eta[\bigoplus_{k=1}^{K} \mathbf{A}^k] \leq \sum_{k=1}^{K} (\eta[\mathbf{A}^k] \cdot \bar{n}_k)$. This bound is achieved iff at most one matrix $\mathbf{A}^k$ contains nonzeros on the diagonal. On the other hand, if all diagonal elements of the matrices $\mathbf{A}^k$ are positive (or all are negative), $\eta[\bigoplus_{k=1}^{K} \mathbf{A}^k] = \sum_{k=1}^{K} (\eta[\mathbf{A}^k] \cdot \bar{n}_k) - (K-1) \cdot n$. As a consequence, the Kronecker sum of $K \geq 2$ matrices (with $n_k > 1$) can never be a full matrix.

We consider a structured model M described as the parallel composition of a set of K submodels $M_{[1,K]}$: each submodel M_k is described as a stochastic automaton in the stochastic automata network (SANs) framework [26], or as a generalized stochastic Petri net (GSPN) in the superposed GSPNs (SGSPNs) framework [17, 21]. The interactions among submodels can be either "simultaneous jumps", when changes of state in two or more submodels must happen in a synchronous manner ("synchronizing events" in SANs or "synchronizing transitions" in SGSPNs), or rate dependencies ("functional rates" in SANs or a simpler type of product-form dependency in SGSPNs [12]). Rate dependencies are discussed in Sect. 5.1. We define

- $\hat{\mathcal{T}}^k$, the set of states of submodel M_k when considered in isolation. $|\hat{\mathcal{T}}^k| = n_k$.

- $\hat{\mathcal{T}} = \hat{\mathcal{T}}^1 \times \ldots \times \hat{\mathcal{T}}^K$, the *potential state space*. $|\hat{\mathcal{T}}| = n_1^K = n$. A state of the composed model is the tuple describing the local state of the K submodels: $i \equiv i_{[1,K]}$.
- $\mathcal{E}_S$, the set of synchronizing events.
- $\mathcal{T}$, the set of reachable, or actual, states in the composed model.

In the presence of synchronizations among submodels, $\hat{\mathcal{T}}$ is often a strict superset of $\mathcal{T}$, which can be efficiently generated from the compositional description of the model. A basic algorithm for state space generation of compositional models is introduced in [22] and further improved in [7] by using an equivalence reduction of the submodel state spaces. The generation of $\mathcal{T}$ containing a few million states can be performed in a matter of minutes, a negligible effort compared to stationary analysis. In [11] an alternative approach using an efficient K-level data-structure is described. From now on, we can then safely assume that $\mathcal{T}$ is available in memory, if needed.

Note that, if $\mathcal{T} \subset \hat{\mathcal{T}}$ and $\mathcal{T}$ has been explored, one might recognize that some local states in $\hat{\mathcal{T}}^k$ are unreachable. We can then define the "actual local" state spaces as the projection of $\mathcal{T}$ on the k-th component:

$$\mathcal{T}^k = \{i_k : \exists j_{[1,K]} \in \mathcal{T}, j_k = i_k\} \subseteq \hat{\mathcal{T}}^k,$$

redefine $\hat{\mathcal{T}}^k$ as $\mathcal{T}^k$, and assume from now on that the two are identical. This improves both memory requirements and execution time, at the cost of requiring the exploration of $\mathcal{T}$.

The compositional definition of M allows a structured description of the transition rate matrix based on the following "local" matrices:

- $\mathbf{W}^k(e)$, a $n_k \times n_k$ matrix describing the effect of synchronizing event e on submodel M_k.
- $\mathbf{R}^k$, a $n_k \times n_k$ matrix describing the effect on submodel M_k of the events local to it.

Using related frameworks, [4, 12, 17, 25, 26] have shown that both the transition rate matrix $\mathbf{R}$ and the infinitesimal generator $\mathbf{Q}$ underlying M can be expressed as the restrictions to the reachable states of appropriate matrices, $\mathbf{R} = \hat{\mathbf{R}}_{\mathcal{T},\mathcal{T}}$ and $\mathbf{Q} = \hat{\mathbf{Q}}_{\mathcal{T},\mathcal{T}}$, defined as Kronecker expressions on the $\mathbf{W}^k(e)$ and $\mathbf{R}^k$ matrices. The expression for $\hat{\mathbf{R}}$ is:

$$\hat{\mathbf{R}} = \underbrace{\sum_{e \in \mathcal{E}_S} rate(e) \cdot \bigotimes_{k=1}^{K} \mathbf{W}^k(e)}_{\text{synchronizing events}} + \underbrace{\bigoplus_{k=1}^{K} \mathbf{R}^k}_{\text{local events}} \quad , \tag{3}$$

where $rate(e)$ is a constant for a given synchronizing event e, and the $\mathbf{W}^k(e)$ and $\mathbf{R}^k$ matrices are either real constants or real functions of the global state. For now we restrict ourselves to constant values for the matrix entries; the case of functional rates is discussed in Sect. 5.1. We observe that the matrices involved in the Kronecker expression of Eq. 3 can be extremely sparse in practice. For example, if the synchronizing event e can occur only in a single local state for submodel M_k and has a deterministic effect on it, $\mathbf{W}^k(e)$ contains exactly one nonzero.

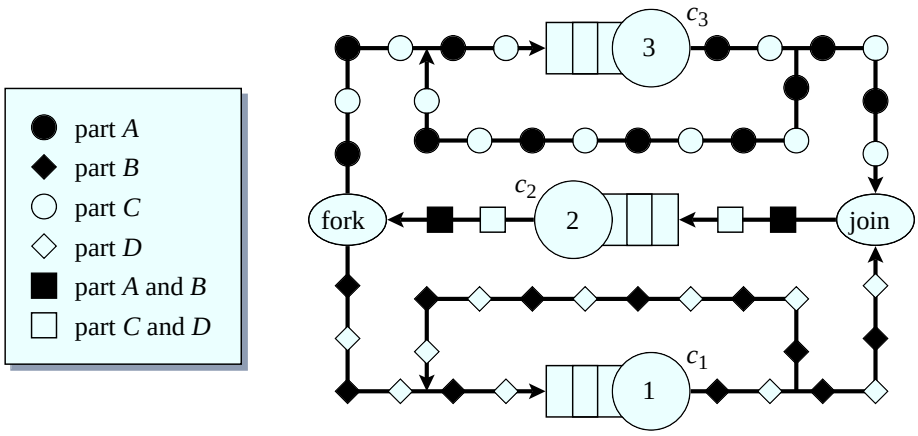

Figure 1: Multiclass Queueing Network for our running example

The expression for $\hat{\mathbf{Q}}$ is analogous to that for $\hat{\mathbf{R}}$ but we omit it because we choose to store $\hat{\mathbf{R}}$ in Kronecker form and the expected holding times $\mathbf{h}$, or the "potential" version $\hat{\mathbf{h}}$, explicitly as a full vector (of course, then, $\mathbf{Q} = \mathbf{R} - diag(\mathbf{h})^{-1}$). Alternatively, we could save memory by using a Kronecker description for the diagonal of $\mathbf{Q}$, at the cost of additional execution time.

2.1 A running example

We now describe the running example used to obtain timing results. It models a flexible manufacturing system (FMS) with three machine centers (c_1, c_2, and c_3) and four types of parts A, B, C, and D being processed. Fig. 1 depicts it as a fork-join queuing network, with machines as queues and parts as customer classes (chains). We assume exponentially distributed service times. Machine c_3 can process up to three parts in parallel, machine c_2 up to two, and machine c_1 only one. Parts A and B are processed with higher priority than C and D.

A part of type A accesses c_3 and, after service completion, it is either rescheduled for processing at c_3 or joined with a part of type B for processing at c_2. A part of type B accesses c_1 and, after service completion, it is either rescheduled for processing at c_1 or joined with a part of type A for processing at c_2. The joint processing of parts of type A and B occurs on c_2 and, after service completion, it yields a product which is delivered and replaced by its original raw parts, to keep a constant stock of material in the system.

The FMS also produces a second product with low priority, to reduce idle time on machines. The low-priority product is processed in the same manner as the high-priority product, but from parts of type C (instead of A) and D (instead of B). The only difference is that processing of the corresponding parts can only take place on a machine that has no high-priority work to be performed (we assume a preemptive priority policy). The parameters n_A, n_B, n_C, and n_D give the number of parts of each type present in the system.

We start with decomposing the model into two submodels according to the priority of parts. Submodel H describes machines c_1, c_2, and c_3 and their processing of the high-priority parts A

Type of matrix	e	$k = H$	$k = L$
$\mathbf{W}^k(e)$	low_1	1,158	584
	low_2	2,259	135
	low_3	2,214	584
$\mathbf{R}^k$	local	11,844	1,438

Table 2: Number of nonzeros using the first decomposition.

and B, including their joint processing. Submodel L considers only low-priority parts C and D, and their joint processing. Since H already models the machines, L must "borrow" the available machine capacities from H. This is done via the synchronizing events $\mathcal{E}_S = \{low_1, low_2, low_3\}$ which are enabled if the state of H shows free capacities at the corresponding machine c_1, c_2, or c_3. As soon as a local event in H leads to a state where high-priority parts require the entire capacity of machine c_i currently serving low-priority parts, the synchronizing event low_i becomes disabled by model H.

For $n_A = n_B = 4$, and $n_C = n_D = 3$ we obtain $|\mathcal{T}^H| = 2{,}394$, $|\mathcal{T}^L| = 652$, and $|\hat{\mathcal{T}}| = |\mathcal{T}| = 1{,}560{,}888$. Table 2 gives the number of nonzeros for the matrices involved in the Kronecker description of $\mathbf{R}$. Matrices are rather sparse, with an average number of nonzeros per row between 0.2 and 0.95. The Kronecker representation uses a total of 20,216 nonzeros, compared to the 13,439,073 nonzeros of an explicit representation.

If matrix entries are stored in double precision, the Kronecker description of $\mathbf{R}$ requires 388,800 bytes. The explicit sparse-storage representation for $\mathbf{R}$ would instead require about 126 MB in single precision or 180 MB in double precision. Obviously, the Kronecker representation of $\mathbf{R}$ is extremely space-efficient in this case.

3 Multiplication using the potential state space

If $\mathbf{A}$ is a $n \times n$ matrix stored explicitly using sparse storage, the complexity of computing the product $\mathbf{x} \cdot \mathbf{A}$ is $O(\eta[\mathbf{A}])$. Storing $\mathbf{A}$ in a full two-dimensional data structure is inefficient for the type of problems we consider; in any case, it is equivalent to assuming that $\eta[\mathbf{A}] = n^2$ from a complexity point of view, so we restrict ourselves to sparse storage from now on. If $\mathbf{A}$ is instead stored implicitly as the Kronecker product of K matrices $\mathbf{A}^k \in \mathbb{R}^{n_k \times n_k}, k \in \{1, \ldots, K\}$, also stored in sparse row-wise or column-wise format, as appropriate, a direct application of Eq. 2 requires $K - 1$ multiplications to obtain each matrix entry. If each $\mathbf{A}_{i,j}$ is computed only once for each pair (i, j) and only nonzero $\mathbf{A}_{i,j}$ entries are computed, the complexity of computing $\mathbf{x} \cdot \mathbf{A}$ becomes $O(K \cdot \eta[\mathbf{A}])$. One of our goals is to improve on this complexity. Sect. 3.1 recalls the shuffle algorithm, which achieves a better complexity by exploiting the inherent structure of a Kronecker product. Sect. 3.2 and 3.3 present instead new algorithms that reduce the complexity by recognizing the presence of common factors in the products making up the entries of $\mathbf{A}$. A

prefix "*Pot-*"indicates that an algorithm works with the potential state space.

In our particular application, the matrices $\mathbf{A}^k$ involved in a Kronecker product are the K matrices $\mathbf{W}^k(e)$ for a given $e \in \mathcal{E}$. Hence, we consider three levels of sparsity, according to the average number $\alpha = \eta[\mathbf{A}^k]/n_k$ of nonzeros per row or column in the matrices $\mathbf{A}^k$ (in the following we assume the same α for all matrices $\mathbf{A}^k$, hence $\eta[\mathbf{A}] = \eta[\otimes \mathbf{A}^k] = n \cdot \alpha^K$):

hypersparse: $\alpha \ll 1 \Rightarrow \eta[\mathbf{A}] \ll n$ (only a few nonzeros; most rows and columns are empty).

ultrasparse: $\alpha \approx 1 \Rightarrow \eta[\mathbf{A}] \approx n$ (each row or column has one nonzero, on average).

sparse: $\alpha \gg 1 \Rightarrow \eta[\mathbf{A}] \gg n$ (any other sparse matrix).

We focus on the case of sparse or ultrasparse, that is, we assume that $\eta[\mathbf{A}^k] \geq n_k$, for all $k = 1, \ldots, K$, but truly hypersparse matrices can occur in the Kronecker approach. Extremely sparse cases might be best managed by explicitly storing a list of triplets $(i, j, \mathbf{A}_{i,j})$, one for each nonzero in $\mathbf{A}$.

3.1 The shuffle algorithm

The first algorithm for the analysis of structured Markov chains was presented in [25]. Fig. 2 shows algorithms *Pot-Sh*, to compute $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{x}} \cdot \otimes_{k=1}^{K} \mathbf{A}^k$, and *Pot-Sh*$^+$, to compute $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{x}} \cdot \mathbf{I}_{n_1^{k-1}} \otimes \mathbf{A}^k \otimes \mathbf{I}_{n_{k+1}^K}$ (from now on, the suffix "+" denotes the version for the simpler case of a product where all matrices are the identity except one).

Pot-Sh considers the matrices $\mathbf{A}^k$ sequentially, exploiting the equality [13]:

$$\bigotimes_{k=1}^{K} \mathbf{A}^k = \prod_{k=1}^{K} \mathbf{S}^T_{(n_1^k, n_{k+1}^K)} \cdot (\mathbf{I}_{\bar{n}_k} \otimes \mathbf{A}^k) \cdot \mathbf{S}_{(n_1^k, n_{k+1}^K)}, \tag{4}$$

where $\mathbf{S}_{(a,b)} \in \{0,1\}^{a \cdot b \times a \cdot b}$ is the matrix describing an (a, b) perfect shuffle permutation:

$$(\mathbf{S}_{(a,b)})_{i,j} = \begin{cases} 1 & \text{if } j = (i \bmod a) \cdot b + (i \operatorname{div} a) \\ 0 & \text{otherwise} \end{cases} .$$

Therefore, a vector-matrix multiplication can be performed using K vector permutations and K multiplications of the type $\mathbf{x} \cdot (\mathbf{I}_{\bar{n}_k} \otimes \mathbf{A}^k)$. Matrix $\mathbf{I}_{\bar{n}_k} \otimes \mathbf{A}^k$ has a peculiar structure: it is simply matrix $\mathbf{A}^k$ repeated $\bar{n}_k$ times over the diagonal, hence, the cost of the k-th multiplication is $O(\bar{n}_k \cdot \eta[\mathbf{A}^k])$, while the permutation costs can be neglected, since they can be incorporated into the algorithm.

As an example, Fig. 3 illustrates the computation of $\mathbf{y} \leftarrow \mathbf{x} \cdot (\mathbf{A} \otimes \mathbf{B})$, following in particular the operations performed to obtain the entry $\mathbf{y}_2$ (follow the entries marked with a diamond). As $\mathbf{S}_{6,1}$ and $\mathbf{S}^T_{6,1}$ are identity matrices, they are omitted from the figure.

The shuffle permutation is encoded in steps 9–12 and 14–17 of *Pot-Sh*. Due to the Kronecker structure, we can perform *Pot-Sh* and *Pot-Sh*$^+$ using a single vector that holds $\hat{\mathbf{x}}$ at the beginning

Pot-Sh(in: $n_{[1,K]}$, $\mathbf{A}^{[1,K]}$; inout: $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$);

```
1.  n_left ← 1;
2.  n_right ← n_2^K;
3.  for k = 1 to K
4.    base ← 0;
5.    jump ← n_k · n_right;
6.    if A^k ≠ I then
7.      for block = 0 to n_left − 1
8.        for offset = 0 to n_right − 1
9.          index ← base + offset;
10.         for h = 0 to n_k − 1
11.           z_h ← x̂_index;
12.           index ← index + n_right;
13.         z' ← z · A^k;
14.         index ← base + offset;
15.         for h = 0 to n_k − 1
16.           ŷ_index ← z'_h;
17.           index ← index + n_right;
18.       base ← base + jump;
19.     x̂ ← ŷ;
20.   n_left ← n_left · n_k;
21.   n_right ← n_right/n_{k+1};      •Let n_{K+1} be 1
```

Pot-Sh$^+$(in: $\mathbf{x}$, n_{left}, n_k, n_{right}, $\mathbf{A}^k$; inout: $\mathbf{y}$);

```
1.  base ← 0;
2.  jump ← n_k · n_right;
3.  for block = 0 to n_left − 1
4.    for offset = 0 to n_right − 1
5.      index ← base + offset;
6.      for h = 0 to n_k − 1
7.        z_h ← x̂_index;
8.        index ← index + n_right;
9.      z' ← z · A^k;
10.     index ← base + offset;
11.     for h = 0 to n_k − 1
12.       ŷ_index ← ŷ_index + z'_h;
13.       index ← index + n_right;
14.   base ← base + jump;
```

Figure 2: Vector-matrix multiplication using perfect shuffles.

and $\hat{\mathbf{y}}$ at the end (provided $\hat{\mathbf{x}}$ is not used elsewhere), since the algorithms overwrite only those values of $\hat{\mathbf{x}}$ which are not used any further or are currently stored in $\mathbf{z}$.

The complexity of *Pot-Sh* [5, 19] can be rewritten as:

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \eta[\mathbf{A}^k]\right) = O\left(n \cdot \sum_{k=1}^{K} \frac{\eta[\mathbf{A}^k]}{n_k}\right) = O\left(n \cdot K \cdot \alpha\right).$$

Hence, *Pot-Sh* is faster than a multiplication using explicit storage iff

$$n \cdot K \cdot \alpha < n \cdot \alpha^K \quad \Longleftrightarrow \quad \alpha > K^{\frac{1}{K-1}}.$$

Fig. 4 illustrates the regions where *Pot-Sh* or ordinary multiplication perform better, according to the values of K (which is small in practical modeling applications) and α. Note that ordinary multiplication is always advantageous in the ultrasparse case.

Pot-Sh$^+$ in Fig. 2 is the specialization of *Pot-Sh* when $\mathbf{A}^k \neq \mathbf{I}$ is true for exactly one k, and it is called with its parameters n_{left} and n_{right} set to n_1^{k-1} and n_{k+1}^K, respectively. Its complexity is

$$O\left(\bar{n}_k \cdot \eta[\mathbf{A}^k]\right) = O\left(n \cdot \alpha\right)$$

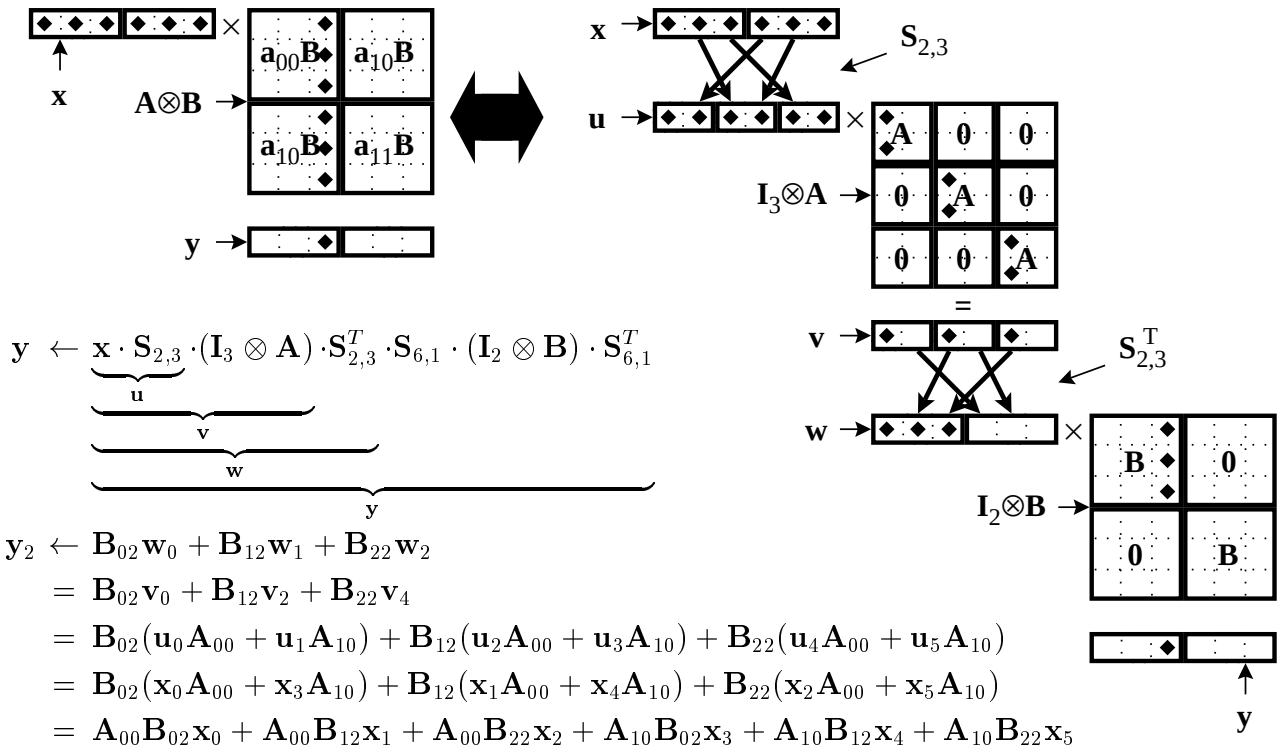

Figure 3: Illustration of the shuffle algorithm.

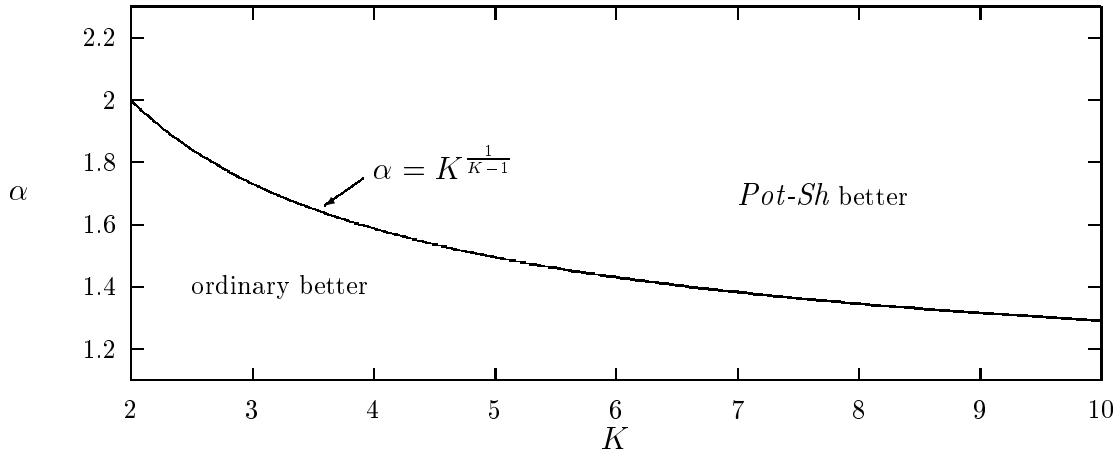

Figure 4: Comparing *Pot-Sh* with ordinary multiplication in the (K, α) plane.

and the resulting complexity of computing $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}} \cdot \bigoplus_{k=1}^{K} \mathbf{A}^k$ using *Pot-Sh*$^+$ is then

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \eta[\mathbf{A}^k]\right) = O\left(n \sum_{k=1}^{K} \frac{\eta[\mathbf{A}^k]}{n_k}\right) = O\left(n \cdot K \cdot \alpha\right).$$

Since the term $\sum_{k=1}^{K} \bar{n}_k \cdot \eta[\mathbf{A}^k]$ is un upper bound on the number of nonzeros in $\mathbf{A} = \bigoplus_{k=1}^{K} \mathbf{A}^k$, using *Pot-Sh*$^+$ saves space, but not time, with respect to the ordinary multiplication algorithm where $\mathbf{A}$ is stored explicitly in sparse form.

Pot-RwEl(in: $i_{[1,K]}, x, n_{[1,K]}, \mathbf{A}^{[1,K]}$; inout: $\hat{\mathbf{y}}$)

1. for each j_1 s.t. $\mathbf{A}^1_{i_1,j_1} > 0$
2. $\quad j'_1 \leftarrow j_1;$
3. $\quad a_1 \leftarrow \mathbf{A}^1_{i_1,j_1};$
4. $\quad$ for each j_2 s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
5. $\quad\quad j'_2 \leftarrow j'_1 \cdot n_2 + j_2;$
6. $\quad\quad a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2};$

$\quad\quad \ldots$

7. $\quad\quad$ for each j_K s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
8. $\quad\quad\quad j'_K \leftarrow j'_{K-1} \cdot n_K + j_K;$
9. $\quad\quad\quad a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K};$
10. $\quad\quad\quad \hat{\mathbf{y}}_{j'_K} \leftarrow \hat{\mathbf{y}}_{j'_K} + x \cdot a_K;$

Pot-Rw(in: $\hat{\mathbf{x}}, n_{[1,K]}, \mathbf{A}^{[1,K]}$; inout: $\hat{\mathbf{y}}$)

1. for $i \equiv i_{[1,K]} = 0$ to $n-1$
2. $\quad$ *Pot-RwEl*$(i, \hat{\mathbf{x}}_i, n_{[1,K]}, \mathbf{A}^{[1,K]}, \hat{\mathbf{y}})$;

Pot-RwEl$^+$(in: $n_k, n^K_{k+1}, i^-_k, i_k, i^+_k, x, \mathbf{A}^k$; inout: $\hat{\mathbf{y}}$)

1. for each j_k s.t. $\mathbf{A}^k_{i_k,j_k} > 0$
2. $\quad j' \leftarrow (i^-_k \cdot n_k + j_k) \cdot n^K_{k+1} + i^+_k;$
3. $\quad \hat{\mathbf{y}}_{j'} \leftarrow \hat{\mathbf{y}}_{j'} + x \cdot \mathbf{A}^k_{i_k,j_k};$

Pot-Rw$^+$(in: $\hat{\mathbf{x}}, n^{k-1}_1, n_k, n^K_{k+1}, \mathbf{A}^k$; inout: $\hat{\mathbf{y}}$)

1. for $i \equiv (i^-_k, i_k, i^+_k) = 0$ to $n-1$
2. $\quad$ *Pot-RwEl*$^+(n_k, n^K_{k+1}, i^-_k, i_k, i^+_k, \hat{\mathbf{x}}_i, \mathbf{A}^k, \hat{\mathbf{y}})$;

Pot-RwCl(in: $\hat{\mathbf{x}}, n_{[1,K]}, \mathbf{A}^{[1,K]}$; inout: $\hat{\mathbf{y}}$)

1. for $i_1 = 0$ to $n_1 - 1$
2. $\quad$ for each j_1 s.t. $\mathbf{A}^1_{i_1,j_1} > 0$
3. $\quad\quad j'_1 \leftarrow j_1;$
4. $\quad\quad a_1 \leftarrow \mathbf{A}^1_{i_1,j_1};$
5. $\quad\quad$ for $i_2 = 0$ to $n_2 - 1$
6. $\quad\quad\quad$ for each j_2 s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
7. $\quad\quad\quad\quad j'_2 \leftarrow j'_1 \cdot n_2 + j_2;$
8. $\quad\quad\quad\quad a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2};$

$\quad\quad\quad \ldots$

9. $\quad\quad\quad\quad$ for $i_K = 0$ to $n_K - 1$
10. $\quad\quad\quad\quad\quad$ for each j_K s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
11. $\quad\quad\quad\quad\quad\quad j'_K \leftarrow j'_{K-1} \cdot n_K + j_K;$
12. $\quad\quad\quad\quad\quad\quad a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K};$
13. $\quad\quad\quad\quad\quad\quad \hat{\mathbf{y}}_{j'_K} \leftarrow \hat{\mathbf{y}}_{j'_K} + \hat{\mathbf{x}}_{i_{[1,K]}} \cdot a_K;$

Pot-RwCl$^+$(in: $\hat{\mathbf{x}}, n^{k-1}_1, n_k, n^K_{k+1}, \mathbf{A}^k$; inout: $\hat{\mathbf{y}}$)

1. for $i^-_k = 0$ to $n^{k-1}_1 - 1$
2. $\quad$ for $i_k = 0$ to $n_k - 1$
3. $\quad\quad$ for each j_k s.t. $\mathbf{A}^k_{i_k,j_k} > 0$
4. $\quad\quad\quad j'_k \leftarrow i^-_k \cdot n_k + j_k;$
5. $\quad\quad\quad$ for $i^+_k = 0$ to $n^K_{k+1} - 1$
6. $\quad\quad\quad\quad j'_K \leftarrow j'_k \cdot n^K_{k+1} + i^+_k;$
7. $\quad\quad\quad\quad \hat{\mathbf{y}}_{j'_K} \leftarrow \hat{\mathbf{y}}_{j'_K} + \hat{\mathbf{x}}_{(i^-_k,i_k,i^+_k)} \cdot \mathbf{A}^k_{i_k,j_k};$

Figure 5: Vector-matrix multiplication by rows (left) or by squares (right).

3.2 A straightforward algorithm using sparse storage

Another approach, close to the definition of Kronecker product of Eq. (2), is given by algorithm *Pot-Rw* in Fig. 5 on the left, which performs the computation $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}} \cdot \mathbf{A}$ and requires sparse row-wise format for the matrices $\mathbf{A}^k$.

Procedure *Pot-RwEl* computes the contribution of a single entry $\hat{\mathbf{x}}_i$ to all the entries of $\hat{\mathbf{y}}$, as $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}}_i \cdot \mathbf{A}_{i,\hat{\jmath}}$. The K nested for-loops in procedure *Pot-RwEl* are required to compute element $\mathbf{A}_{i,j}$. Some of the computation needed to obtain $\mathbf{A}_{i,j}$ is reused for other elements of matrix $\mathbf{A}$ on the same row. On a given call to *Pot-RwEl*, statement $a_k \leftarrow a_{k-1} \cdot \mathbf{A}^k_{i_k,j_k}$ is reached $\prod_{h=1}^{k} \eta[\mathbf{A}^h_{i_h,\mathcal{T}^h}] = O(\alpha^k)$ times. *Pot-Rw* calls *Pot-RwEl* n times, hence its complexity is

$$O\left(n \cdot \sum_{k=1}^{K} \alpha^k\right) = \begin{cases} O\left(n \cdot K\right) = O(K \cdot \eta[\mathbf{A}]) & \text{ultrasparse} \\ O(n \cdot \alpha^K) = O(\eta[\mathbf{A}]) & \text{sparse} \end{cases} \tag{5}$$

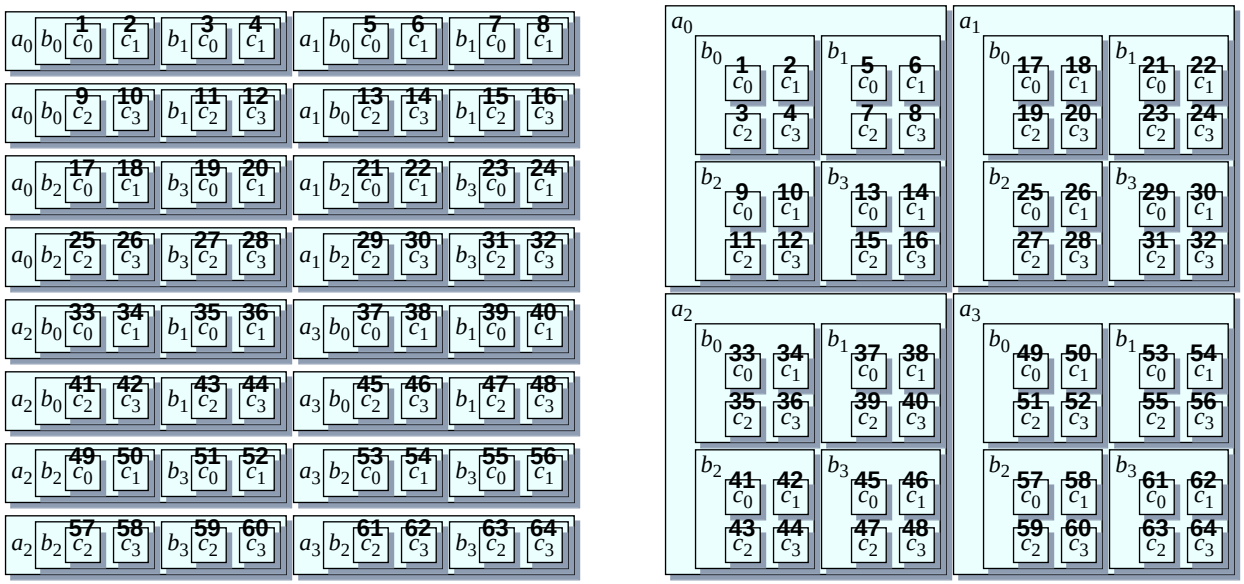

Figure 6: Multiplication by rows (left) and by squares (right).

In other words, the multiplications needed to compute a_2 through a_{K-1} are effectively amortized only if $\alpha \gg 1$.

The analogous algorithm *Pot-Cl* and procedure *Pot-ClEl*, for multiplication by columns, are omitted. Each call to *Pot-ClEl* computes a single entry $\hat{\mathbf{y}}_j$ of $\hat{\mathbf{y}}$ as the inner product $\hat{\mathbf{x}} \cdot \mathbf{A}_{\hat{\mathcal{T}},j}$, where $j \equiv j_{[1,K]}$ in our mixed-base notation. *Pot-Cl* has the same complexity as *Pot-Rw* but requires sparse column-wise storage for the matrices $\mathbf{A}^k$. However, *Pot-Rw* and *Pot-Cl* differ in one important aspect: the multiplication performed by *Pot-ClEl* requires only one scalar accumulator, while *Pot-RwEl* uses the vector $\hat{\mathbf{y}}$ itself as an accumulator. If we follow the good numerical practice of using higher precision for accumulators, *Pot-Rw* has larger memory requirements than *Pot-Cl*.

The simplified multiplication algorithm *Pot-Rw*$^+$, used to compute Kronecker sums, is also shown in Fig. 5 on the left. Its complexity is

$$O\left(n \cdot \frac{\eta[\mathbf{A}^k]}{n_k}\right) = O\left(n \cdot \alpha\right).$$

The resulting complexity of computing $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}} \cdot \bigoplus_{k=1}^{K} \mathbf{A}^k$ using *Pot-Rw*$^+$ is then

$$O\left(n \cdot \sum_{k=1}^{K} \frac{\eta[\mathbf{A}^k]}{n_k}\right) = O\left(n \cdot K \cdot \alpha\right).$$

Again, as for shuffle, there is no gain in time over the ordinary vector-matrix multiplication algorithm.

3.3 Interleaving rows and columns

Fig. 6 on the left illustrates graphically the number of operations and the order in which elements are accessed by algorithm *Pot-Rw* (*Pot-Cl* is analogous), when computing the product

$$\hat{\mathbf{x}} \cdot \mathbf{A} = \hat{\mathbf{x}} \cdot \left(\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \otimes \begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \end{bmatrix} \otimes \begin{bmatrix} c_0 & c_1 \\ c_2 & c_3 \end{bmatrix} \right).$$

The resulting matrix $\mathbf{A}$ contains $8 \times 8 = 64$ entries, each of which is the product of three real numbers $a_i b_j c_l$, $0 \leq i, j, l \leq 3$. Since the product of three numbers requires two multiplications, we could trivially compute all the entries of $\mathbf{A}$ by performing $64 \times 2 = 128$ multiplications. Algorithm *Pot-Rw* reduces this complexity by exploiting some level of factoring. In Fig. 6 on the left, for example, the product $a_0 \cdot b_0$ appears in both $a_0 \cdot b_0 \cdot c_0$ and $a_0 \cdot b_0 \cdot c_1$, and is computed only once. Hence, the overall number of multiplications performed by *Pot-Rw* to compute all the entries of $\mathbf{A}$ equals the number of "*b*" and "*c*" boxes in the figure, $64 + 32 = 96$. The boldface numbers indicate the order in which the entries of $\mathbf{A}$ are generated.

However, neither *Pot-Rw* nor *Pot-Cl* fully exploit the existing common factors in the entries of $\mathbf{A}$. To do so, we must use algorithm *Pot-RwCl* instead, shown in Fig. 5 on the right. The idea is to examine the entries of $\mathbf{A}$ not in row or column order, but by interleaving the indices (i_1, j_1, i_2, j_2, etc.), that is, "by squares". Note that the case $\mathbf{A}^k = \mathbf{I}$ can be recognized and optimized to avoid unnecessary multiplications and complex index transformations.

Fig. 6 on the right shows how this works on our small 8×8 example. The number of operations (of "*b*" and "*c*" boxes) is now $64 + 16 = 80$, and it is apparent how *Pot-RwCl* can substantially outperform *Pot-Rw* and *Pot-Cl* as the number K and the size of the matrices involved increases. However, considering again the boldface numbers that indicate the order in which the entries of $\mathbf{A}$ are generated, one can see how *Pot-RwCl* does not accommodate an algorithm requiring access strictly by-rows or by-columns. Indeed, the name *Pot-RwCl* stresses that the first for-loop is on i_1, a row index, the second one is on j_1, a column index, and so on. An analogous *Pot-ClRw* algorithm could be defined, with for-loops over the indices j_1, i_1, j_2, i_2, etc., but it would still have a complex access pattern, neither by rows nor by columns.

To obtain the complexity of *Pot-RwCl*, we can observe that statement $a_k \leftarrow a_{k-1} \cdot \mathbf{A}^k_{i_k, j_k}$ is reached $\prod_{l=1}^{k} \eta[\mathbf{A}^l]$ times, hence the overall complexity is

$$O\left(\sum_{k=1}^{K} \prod_{l=1}^{k} \eta[\mathbf{A}^l] \right) = O\left(\prod_{l=1}^{K} \eta[\mathbf{A}^l] \right) = O\left(n \cdot \alpha^K \right),$$

that is, the same as that of ordinary multiplication, regardless of whether the matrices involved are sparse or just ultrasparse.

Just as for *Pot-Rw* and *Pot-Cl*, a specialized version of *Pot-RwCl* can be defined, to be used when dealing with Kronecker sums (see algorithm *Pot-RwCl*$^+$ in Fig. 5). Its complexity is the same as that of *Pot-Sh*, *Pot-Rw*, or *Pot-Cl*. Our running example is used for a comparison of these algorithms, in Section 5.

Type of matrix	e	$k=1$	$k=2$	$k=3$	$k=4$
$\mathbf{W}^k(e)$	low_3	120	—	42	—
	low_2	105	—	21	—
	low_1	—	35	—	30
	$join_{AB}$	56	35	—	—
	$fork_{AB}$	56	35	—	—
	$join_{CD}$	—	—	21	15
	$fork_{CD}$	—	—	21	15
$\mathbf{R}^k$	local	280	140	42	30

Table 3: Number of nonzeros using the second decomposition.

4 Multiplication using the actual state space

The decomposition considered so far for our running example satisfies $\hat{\mathcal{T}} = \mathcal{T}$. However, a non-trivial decomposition with $\hat{\mathcal{T}} = \mathcal{T}$ might not exist for a given model or might be unknown to a modeler. Due to the space limitations we do not consider a different model where $|\hat{\mathcal{T}}| \gg |\mathcal{T}|$; instead, we simply use a second decomposition that further refines the submodels H and L of our running example. For the high-priority parts, we define submodel 1, describing the processing of parts of type A and their joint processing with parts of type B, and submodel 2, describing the processing of parts of type B on machine c_1. For the low-priority parts, we define analogous submodels 3 and 4.

The synchronizing events are then

$$\mathcal{E}_S = \{low_1, low_2, low_3, join_{AB}, fork_{AB}, join_{CD}, fork_{CD}\}$$

where the "join" and "fork" events correspond to the start and end of assembly for parts A and B, or C and D, respectively.

For $n_A = n_B = 4$, and $n_C = n_D = 3$, the cardinalities of the local state spaces are $|\mathcal{T}^1| = 126$, $|\mathcal{T}^2| = 70$, $|\mathcal{T}^3| = 56$, and $|\mathcal{T}^4| = 35$. The potential state space is now much larger, $|\hat{\mathcal{T}}| =$ 17,287,200, while $|\mathcal{T}| =$ 1,560,888, as before, since we are modeling the same system. Table 3 gives the number of nonzeros for the matrices involved in the Kronecker description of $\mathbf{R}$ (missing entries indicate identity matrices, which do not need to be stored explicitly).

The matrices for the Kronecker description of $\mathbf{R}$ now use a truly negligible amount of memory, 29,148 bytes, but *Pot-Sh*, *Pot-Rw*, and *Pot-Cl* need a large amount of space to allocate vectors of length $\hat{\mathcal{T}}$, even if we are really interested only in the elements corresponding to $\mathcal{T}$.

When the difference between $\hat{\mathcal{T}}$ and $\mathcal{T}$ is large, the algorithms of the previous section pay additional costs in terms of space, but also in terms of time, since unnecessary multiplications are performed for unreachable states [19]. Thus, we now consider methods that compute $\hat{\mathbf{y}}_{\mathcal{T}} = \hat{\mathbf{y}}_{\mathcal{T}} + \hat{\mathbf{x}}_{\mathcal{T}} \cdot \mathbf{A}_{\mathcal{T},\mathcal{T}}$, where $\mathcal{T} \subseteq \hat{\mathcal{T}}$ is the actual state space and $\hat{\mathbf{y}}_{\mathcal{T}}$ and $\hat{\mathbf{x}}_{\mathcal{T}}$ are stored using arrays $\mathbf{y}$

Act-RwEl$_1$(in: $i_{[1,K]}, x, \mathbf{A}^{[1,K]}$; inout: $\mathbf{y}$)

1. for each j_1 s.t. $\mathbf{A}^1_{i_1,j_1} > 0$
2. $\quad a_1 \leftarrow \mathbf{A}^1_{i_1,j_1}$;
3. $\quad$ for each j_2 s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
4. $\quad\quad a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2}$;

$\quad\quad\quad \ldots$

5. $\quad\quad\quad$ for each j_K s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
6. $\quad\quad\quad\quad a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K}$;
7. $\quad\quad\quad\quad J \leftarrow \Psi(j_{[1,K]})$;
8. $\quad\quad\quad\quad \mathbf{y}_J \leftarrow \mathbf{y}_J + x \cdot a_K$;

Act-Rw$_1$(in: $\mathbf{x}, \mathbf{A}^{[1,K]}, \mathcal{T}$; inout: $\mathbf{y}$)

1. for each $i_{[1,K]} \in \mathcal{T}$
2. $\quad I \leftarrow \Psi(i_{[1,K]})$;
3. $\quad$ *Act-RwEl*$_1(i_{[1,K]}, \mathbf{x}_I, \mathbf{A}^{[1,K]}, \mathbf{y})$;

Figure 7: Vector-matrix multiplication by rows for a subset $\mathcal{T}$ of the states.

and $\mathbf{x}$, of size $|\mathcal{T}|$. Specifically, for $i \in \mathcal{T}$, $\hat{\mathbf{x}}_i$ is stored in position $I = \Psi(i)$ of $\mathbf{x}$ (i.e., $\hat{\mathbf{x}}_i = \mathbf{x}_I$), where $\Psi(i)$ counts the number of reachable states that precede i in lexicographic order ($\Psi(i) = \mathsf{null}$ if $i \notin \mathcal{T}$). To focus on the reachable states alone, we need to:

- Generate $\mathcal{T}$. Efficient algorithms for the generation of $\mathcal{T}$ can be found in [11, 22]
- Ensure that only $\mathbf{A}_{\mathcal{T},\mathcal{T}}$ contributes to the value of $\mathbf{y}$. If $\mathbf{A}$ is one of the matrices whose sum constitutes $\hat{\mathbf{R}}$, then $\mathbf{A}_{i,j} = 0$ whenever $i \in \mathcal{T}$ and $j \notin \mathcal{T}$, that is, starting from a reachable state, only other reachable states can be reached. In matrix form, this implies that $\hat{\mathbf{R}}_{\mathcal{T},\hat{\mathcal{T}}\setminus\mathcal{T}} = \mathbf{0}$ [12, 21]. The reverse is unfortunately not true: if $i \notin \mathcal{T}$ and $j \in \mathcal{T}$, $\mathbf{A}_{i,j}$ can be positive, that is, reachable states can be reached from unreachable states.
- Find an efficient way to compute $\Psi : \hat{\mathcal{T}} \to \{0, \ldots, |\mathcal{T}| - 1, \mathsf{null}\}$. We use a logarithmic search in $\mathcal{T}$, and show how the overhead is reduced by using an appropriate data structure to store $\mathcal{T}$. This is one of our main contributions in this paper.

Algorithm *Pot-Sh* is not amenable to this approach since it sequentializes the effect of a synchronizing event according to (4), which results in possibly unreachable intermediate states. During the k-th step of the for-loop in line 3 of *Pot-Sh*, the multiplication of $\mathbf{A}^k$ temporarily assigns values to positions of $\hat{\mathbf{y}}$ where no state change has taken place yet, according to $\mathbf{A}^l, k < l < K$. Hence, the price for a fast shuffle permutation is the need for $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ to be of dimension $|\hat{\mathcal{T}}|$. In an implementation, a single vector suffices to represent $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$.

For a complete iterative solution, other vectors can be reduced to size $|\mathcal{T}|$ using Ψ, and zero entries in $\hat{\mathbf{x}}$ can be skipped to save multiplications (the latter requires a simple test for zero in the inner multiplication $\mathbf{z}' \leftarrow \mathbf{z} \cdot \mathbf{A}^k$, line 13 in *Pot-Sh* and line 9 in *Pot-Sh*$^+$). This results in the *Act-Sh-JCB* solution algorithm in Sect. 6.2.

Algorithm *Act-Rw*$_1$ [21] in Fig. 7 modifies *Pot-Rw*, by accessing only elements corresponding to reachable states. A prefix "*Act-*" indicates that the algorithms works with the actual state space. We omit the algorithm *Act-Rw*$_1^+$ to compute Kronecker sums, since an analogous discussion as for

Pot-Rw$^+$ applies. Line 1 in *Act-Rw*$_1$ selects only elements of $\mathcal{T}$ among those in $\hat{\mathcal{T}}$. This requires no additional overhead as long as the elements of $\mathcal{T}$ can be accessed sequentially according to the order Ψ. The assignment in line 7 of *Act-RwEl*$_1$, however, requires finding the index $J = \Psi(j_{[1,K]})$ of the element $j_{[1,K]}$ in the array $\mathbf{y}$. If the computation of Ψ uses a binary search on a vector representation of Ψ, a multiplicative overhead factor $O(\log|\mathcal{T}|)$ is encountered in the innermost for-loop. The overall complexity of *Act-Rw*$_1$ is then derived analogously to that of *Pot-Rw*. On a given call to *Act-RwEl*$_1$, statement $a_k \leftarrow a_{k-1} \cdot \mathbf{A}^k_{i,k,j_k}$ is reached $\prod_{h=1}^{k} \eta[\mathbf{A}^h_{i_h,\mathcal{T}^h}] = O(\alpha^k)$ times, and statement $J \leftarrow \Psi(j_{[1,K]})$ is reached $\eta[\mathbf{A}_{i_{[1,K]},\mathcal{T}}] = O(\alpha^K)$ times. *Act-Rw*$_1$ calls *Act-RwEl*$_1$ once for each $i_{[1,K]} \in \mathcal{T}$, hence its complexity is

$$O\left(|\mathcal{T}| \cdot \left(\sum_{k=1}^{K} \alpha^k + \alpha^K \cdot \log|\mathcal{T}|\right)\right) = \begin{cases} O\left(|\mathcal{T}| \cdot (K + \log|\mathcal{T}|)\right) & \text{ultrasparse} \\ O\left(|\mathcal{T}| \cdot \alpha^K \cdot \log|\mathcal{T}|\right) & \text{sparse} \end{cases} \tag{6}$$

Since $K < \log|\mathcal{T}|$ in practical modeling situations, we can conclude that *Act-Rw*$_1$ has a $\log|\mathcal{T}|$ overhead with respect to ordinary multiplication, regardless of the matrix sparsity.

In a multiplication by columns, the situation is potentially worse. *Act-Cl*$_1$, the version analogous to *Act-Rw*$_1$, must avoid the "spurious" entries in $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$. In *Act-ClEl*$_1$, the index $I \leftarrow \Psi(i_{[1,K]})$ computed by the binary search returns null if the "from" state $i_{[1,K]}$ is not reachable. Hence, *Act-ClEl*$_1$ must test whether $I =$ null and, if so, ignore entry $\mathbf{A}_{i_{[1,K]},j_{[1,K]}}$. The average cost of these binary searches is slightly higher than for *Act-RwEl*$_1$, since searching a state not in $\mathcal{T}$ represents a worst-case scenario, but, more importantly, the complexity of *Act-Cl*$_1$ must account for all searches performed, regardless of whether they are successful or not. The number of such searches is equal to the number of nonzeros in the columns of $\mathbf{A}$ corresponding to $\mathcal{T}$, $\eta[\mathbf{A}_{\hat{\mathcal{T}},\mathcal{T}}]$, while only $\eta[\mathbf{A}_{\mathcal{T},\mathcal{T}}]$ searches are performed by *Act-Rw*$_1$. The sparser $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$ is, the closer the performance of *Act-Cl*$_1$ is to that of *Act-Rw*$_1$, and the two complexities coincide when $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}} = \mathbf{0}$ (this can happen even when $\hat{\mathcal{T}}$ is a strict superset of $\mathcal{T}$).

4.1 Reducing the $\log|\mathcal{T}|$ overhead

The multiplicative overhead $\log|\mathcal{T}|$ in *Act-Rw*$_1$ and *Act-Cl*$_1$ results from a worst-case assumption that we must search in a set of size $|\mathcal{T}|$ to compute each value of Ψ.

[23] discusses approaches to reduce this overhead, but a more effective solution is obtained by using the multilevel data structure shown in Fig. 8 to store $\mathcal{T}$ [11].

Before explaining this data structure, we introduce the following sets:

- $\hat{\mathcal{T}}_1^k = \mathcal{T}^1 \times \cdots \times \mathcal{T}^k = \{0, \ldots, n_1 - 1\} \times \cdots \times \{0, \ldots, n_k - 1\}$, the result of the projection of the potential state space over the first k components.
- $\mathcal{T}_1^k = \{i_{[1,k]} : \exists i_{[k+1,K]}, i_{[1,K]} \in \mathcal{T}\}$, the projection of the actual state space over the first k components.
- $\mathcal{T}^k(i_{[1,k-1]}) = \{i_k : i_{[1,k]} \in \mathcal{T}_1^k\}$, the local states for M_k that can occur when the local states for M_1 through M_{k-1} are i_1 through i_{k-1}, respectively.

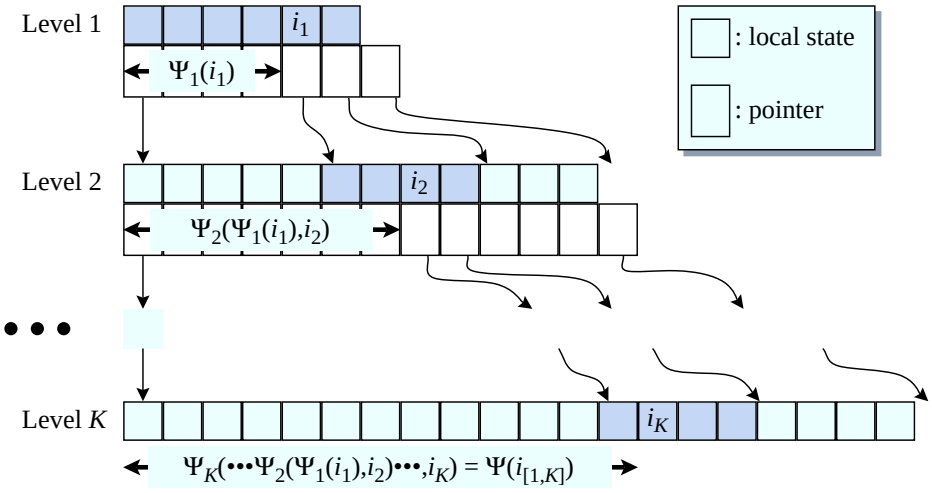

Figure 8: Storage scheme for computation of Ψ.

In particular, $\hat{\mathcal{T}}_1^K \equiv \hat{\mathcal{T}}$, $\mathcal{T}_1^K \equiv \mathcal{T}$, and we can define $\mathcal{T}^1(i_{[1,0]})$ simply as $\mathcal{T}^1$.

In Fig. 8, the elements of the array at level k contain local states for submodel M_k. When searching for a given state $i_{[1,K]}$, we search first for i_1 in the array at level 1, containing $\mathcal{T}^1$. After finding i_1, we follow its pointer to the array at level 2. The greyed-out portion of this array contains $\mathcal{T}^2(i_1)$. We then search for i_2 in this portion, and so on, until we find the local state i_K in the greyed-out portion of the last array, corresponding to $\mathcal{T}^K(i_{[1,K-1]})$. The displacement of this local state in the entire array at level K is $\Psi(i_{[1,K]})$. If, at any level, we are unable to find i_k, we can conclude that the state we are searching is not in $\mathcal{T}$, that is, $\Psi(i_{[1,K]}) = \mathsf{null}$.

The arrays at levels 1 through $K-1$ are usually small compared to the last level and the array at level K, of size $|\mathcal{T}|$, can be compressed into $\lceil \log_2 n_K \rceil \cdot |\mathcal{T}|$ bits. Hence $\mathcal{T}$ can be stored in $O(|\mathcal{T}| \cdot \log_2 n_K)$ bits [11].

For our purposes, however, the real advantage of this data structure is the amortization of the logarithmic searches. For a given $i_{[1,K]}$, we compute $\Psi(i_{[1,K]})$ in K steps:

$$\Psi(i_{[1,K]}) = \Psi_K(\cdots \Psi_2(\Psi_1(i_1), i_2) \cdots, i_K).$$

When searching for a second state $i'_{[1,K]}$ such that $i_{[1,k]} = i'_{[1,k]}$, we can reuse the work in the first k of these steps. In other words, if we saved the pointers identifying the greyed array for $\mathcal{T}^{k+1}(i_{[1,k]})$ at level $k+1$ where we found i_{k+1}, we can now start our search for i'_{k+1} in that portion, instead of starting all over at level 1.

This results in algorithms *Act-Rw*$_2$ and *Act-Cl*$_2$. Fig. 9 shows *Act-Cl*$_2$, since it is used in the Gauss-Seidel type solution algorithm *Act-Cl*$_2$*-GSD* of Sect. 6. The tests for $I_k \neq \mathsf{null}$ for $k < K$ are necessary because an event might be inactive due to a submodel M_h with $h > k$, but there might not be a matching state for $\Psi_k(J_{k-1}, j_k)$ at level k already. This is possible not only for *Act-Cl*$_2$, but also for *Act-Rw*$_2$ (we thank A. S. Miner for providing an example where this occurs). In addition, *Act-Cl*$_2$ is still affected by nonzeros in $\mathbf{A}_{\hat{\mathcal{T}},\mathcal{T}}$, as discussed above in this section, requiring the test $I_K \neq \mathsf{null}$ in the innermost for-loop. The analogous test is instead not needed in *Act-Rw*$_2$.

The complexity of *Act-ClEl*$_2$ is now dominated by the searches at each level, since for each multiplication at level k, a $O(\log n_k)$ search is performed as well. On a given call to *Act-ClEl*$_2$, statement $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$ is reached $\eta\left[\left(\bigotimes_{h=1}^{k}\mathbf{A}^h\right)_{\mathcal{T}_1^{k-1}\times\mathcal{T}^k, j_{[1,k]}}\right] = O(\alpha^k)$ times. *Act-Cl*$_2$ calls *Act-ClEl*$_2$ for each $j_{[1,K]} \in \mathcal{T}$, hence its complexity is

$$O\left(|\mathcal{T}| \cdot \sum_{k=1}^{K} \alpha^k \cdot \log n_k\right) = \begin{cases} O\left(|\mathcal{T}| \cdot \sum_{k=1}^{K} \log n_k\right) = O\left(|\mathcal{T}| \cdot \log n\right) & \text{ultrasparse} \\ O\left(|\mathcal{T}| \cdot \alpha^K \cdot \log n_K\right) & \text{sparse} \end{cases} \tag{7}$$

(the simplification for the sparse case is correct provided that n_K is at least comparable to n_k, for any $k < K$).

The complexity of *Act-Rw*$_2$ is analogous, except that the amount of computation performed in *Act-Cl*$_2$ is still worse than for *Act-Rw*$_2$ because searches for unreachable states are still possible. The problem is less serious than for *Act-Cl*$_1$, though, because entries in $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$ may now be discovered before reaching the innermost for-loop, if $i_{[1,k]} \notin \mathcal{T}_1^k$ for some $k < K$.

Comparing Eq. (7) with Eq. (6), we conclude that *Act-Rw*$_2$ and *Act-Cl*$_2$ have better complexity in the sparse case, since they can effectively amortize the logarithmic searches at each level only when the matrices $\mathbf{A}^k$ have multiple elements per row. However, in the ultrasparse case, they have an overhead of $\log n = \log|\hat{\mathcal{T}}|$ instead of just $\log|\mathcal{T}|$. This is due to the pessimistic assumption that a logarithmic search at level k requires $O(\log n_k)$ comparisons. If, for each k, all sets $\mathcal{T}^k(i_{[1,k-1]})$ were of approximately the same size, then their complexity in the ultrasparse case would be reduced to $O(|\mathcal{T}| \cdot \log|\mathcal{T}|)$, but this might not be the case in practice. One factor not evidenced by the complexity expressions, though, is that, unlike *Act-Rw*$_1$ and *Act-Cl*$_1$, *Act-Rw*$_2$ and *Act-Cl*$_2$ avoid reaching the innermost for-loop whenever a partial state is not reachable, so they might actually perform better than *Act-Rw*$_1$ and *Act-Cl*$_1$ even in the ultrasparse case.

Act-Rw$_2^+$ and *Act-Cl*$_2^+$ are the simplified algorithms for matrices arising from a Kronecker sum. Fig. 9 shows *Act-Cl*$_2^+$. On a given call, *Act-ClEl*$_2^+$ reaches statements $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$ and $I_h \leftarrow \Psi_h(I_{h-1}, j_h)$ for $k < h \le K$, with an overall cost of $O\left(\sum_{h=k}^{K} \log n_h\right)$, at most $\eta[\mathbf{A}_{\mathcal{T}^k, j_k}] = O(\alpha)$ times. *Act-Cl*$_2^+$ calls *Act-ClEl*$_2^+$ once for each $j_{[1,K]} \in \mathcal{T}$, hence its complexity is

$$O\left(|\mathcal{T}| \cdot \alpha \cdot \sum_{h=k}^{K} \log n_h\right).$$

The resulting complexity of computing $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{x} \cdot \left(\bigoplus_{k=1}^{K}\mathbf{A}^k\right)_{\mathcal{T},\mathcal{T}}$ using *Act-Cl*$_2^+$ is then

$$O\left(\sum_{k=1}^{K} |\mathcal{T}| \cdot \alpha \cdot \sum_{h=k}^{K} \log n_h\right) = O\left(|\mathcal{T}| \cdot \alpha \cdot \sum_{k=1}^{K} k \cdot \log n_k\right) = O\left(|\mathcal{T}| \cdot \alpha \cdot K \cdot \log n\right).$$

Thus, the logarithmic search overhead decreases from submodel M_1, where no amortization occurs, to M_K, where only an overhead $\log n_K$ is encountered, but the overall overhead remains $\log n$, since the number of nonzeros in $\bigoplus_{k=1}^{K}\mathbf{A}^k$ is approximately $|\mathcal{T}| \cdot \alpha \cdot K$.

Act-ClEl$_2$(in: $j_{[1,K]}, \mathbf{x}, \mathbf{A}^{[1,K]}$; inout: y)

1. for each i_1 s.t. $\mathbf{A}^1_{i_1,j_1} > 0$
2. $I_1 \leftarrow \Psi_1(i_1)$;
3. if $I_1 \neq$ null then
4. $a_1 \leftarrow \mathbf{A}^1_{i_1,j_1}$;
5. for each i_2 s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
6. $I_2 \leftarrow \Psi_2(I_1, i_2)$;
7. if $I_2 \neq$ null then
8. $a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2}$;

...

9. for each i_K s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
10. $I_K \leftarrow \Psi_K(I_{K-1}, i_K)$;
11. if $I_K \neq$ null then
12. $a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K}$;
13. $y \leftarrow y + \mathbf{x}_{I_K} \cdot a_K$;

Act-Cl$_2$(in: $\mathbf{x}, \mathbf{A}^{[1,K]}, \mathcal{T}$; inout: $\mathbf{y}$)

1. for each $j_{[1,K]} \in \mathcal{T}$
2. $J \leftarrow \Psi(j_{[1,K]})$;
3. *Act-ClEl*$_2(j_{[1,K]}, \mathbf{x}, \mathbf{A}^{[1,K]}, \mathbf{y}_J)$;

Act-RwCl(in: $\mathbf{x}, \mathbf{A}^{[1,K]}, \mathcal{T}$; inout: $\mathbf{y}$)

1. for each $i_1 \in \mathcal{T}^1$
2. $I_1 \leftarrow \Psi_1(i_1)$;
3. for each j_1 s.t. $\mathbf{A}^1_{i_1,j_1} > 0$
4. $J_1 \leftarrow \Psi_1(j_1)$;
5. if $J_1 \neq$ null then
6. $a_1 \leftarrow \mathbf{A}^1_{i_1,j_1}$;
7. for each $i_2 \in \mathcal{T}^2(i_1)$
8. $I_2 \leftarrow \Psi_2(I_1, i_2)$;
9. for each j_2 s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
10. $J_2 \leftarrow \Psi_2(J_1, j_2)$;
11. if $J_2 \neq$ null then
12. $a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2}$;

...

13. for each $i_K \in \mathcal{T}^K(i_{[1,K-1]})$
14. $I_K \leftarrow \Psi_K(I_{K-1}, i_K)$;
15. for each j_K s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
16. $J_K \leftarrow \Psi_K(J_{K-1}, j_K)$;
17. if $J_K \neq$ null then
18. $a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K}$;
19. $\mathbf{y}_{J_K} \leftarrow \mathbf{y}_{J_K} + \mathbf{x}_{I_K} \cdot a_K$;

Act-ClEl$_2^+$(in: $j_{[1,K]}, \mathbf{x}, k, \mathbf{A}^k$; inout: y)

1. $I_{k-1} \leftarrow \Psi_{k-1}(\ldots \Psi_2(\Psi_1(j_1), j_2) \ldots, j_{k-1})$;
2. for each i_k s.t. $\mathbf{A}^k_{i_k,j_k} > 0$
3. $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$;
4. if $I_k \neq$ null then
5. $I_{k+1} \leftarrow \Psi_{k+1}(I_k, j_{k+1})$;
6. if $I_{k+1} \neq$ null then

...

7. $I_K \leftarrow \Psi_K(I_{K-1}, j_K)$;
8. if $I_K \neq$ null then
9. $y \leftarrow y + \mathbf{x}_{I_K} \cdot \mathbf{A}^k_{i_k,j_k}$;

Act-Cl$_2^+$(in: $\mathbf{x}, k, \mathbf{A}^k, \mathcal{T}$; inout: $\mathbf{y}$)

1. for each $j_{[1,K]} \in \mathcal{T}$
2. $J \leftarrow \Psi(j_{[1,K]})$;
3. *Act-ClEl*$_2^+(j_{[1,K]}, \mathbf{x}, k, \mathbf{A}^k, \mathbf{y}_J)$;

Act-RwCl$^+$(in: $\mathbf{x}, k, \mathbf{A}^k, \mathcal{T}$; inout: $\mathbf{y}$)

1. for each $i_{[1,k-1]} \in \mathcal{T}_1^{k-1}$
2. $I_{k-1} \leftarrow \Psi_{k-1}(\ldots \Psi_2(\Psi_1(i_1), i_2) \ldots, i_{k-1})$;
3. for each $i_k \in \mathcal{T}^k(i_{[1,k-1]})$
4. $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$;
5. for each j_k s.t. $\mathbf{A}^k_{i_k,j_k} > 0$
6. $J_k \leftarrow \Psi_k(I_{k-1}, j_k)$;
7. for each $i_{k+1} \in \mathcal{T}^{k+1}(i_{[1,k]})$
8. $I_{k+1} \leftarrow \Psi_{k+1}(I_k, i_{k+1})$;
9. $J_{k+1} \leftarrow \Psi_{k+1}(J_k, i_{k+1})$;

...

10. for each $i_K \in \mathcal{T}^K(i_{[1,K-1]})$
11. $I_K \leftarrow \Psi_K(I_{K-1}, i_K)$;
12. $J_K \leftarrow \Psi_K(J_{K-1}, i_K)$;
13. $\mathbf{y}_{J_K} \leftarrow \mathbf{y}_{J_K} + \mathbf{x}_{I_K} \cdot \mathbf{A}^k_{i_k,j_k}$;

Figure 9: Better vector-matrix multiplication algorithms for a subset of the states.

4.2 Interleaving rows and columns for a subset of the states

Act-Rw$_2$ and *Act-Cl*$_2$ fail to amortize the logarithmic searches in the case of ultrasparse matrices because the K nested for-loops in *Act-RwEl*$_2$ and *Act-ClEl*$_2$ consider only the entries on a given row or column of $\mathbf{A}$, respectively. If $\mathbf{A}$ is ultrasparse, only one entry is found, and no amortization occurs.

To further improve the complexity, we apply the interleaving idea of Sect. 3.3 to the case where $\hat{\mathcal{T}} \supset \mathcal{T}$, resulting in algorithm *Act-RwCl*, shown in Fig. 9 on the right. The statements $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$ do not require a search, since all states $i_k \in \mathcal{T}^k(i_{[1,k-1]})$ are enumerated sequentially in the for-loop. A search is performed only to obtain $J_k \leftarrow \Psi_k(J_{k-1}, j_k)$ and the tests $J_k \neq \mathsf{null}$ are necessary, as already discussed for *Act-Rw*$_2$ and *Act-Cl*$_2$.

Statement $J_k \leftarrow \Psi_k(J_{k-1}, j_k)$ is performed $\eta\left[\left(\bigotimes_{h=1}^{k} \mathbf{A}^h\right)_{\mathcal{T}_1^k, \mathcal{T}_1^{k-1} \times \mathcal{T}^k}\right] = O(|\mathcal{T}_1^k| \cdot \alpha^k)$ times, hence the complexity of *Act-RwCl* is:

$$O\left(\sum_{k=1}^{K} |\mathcal{T}_1^k| \cdot \alpha^k \cdot \log n_k\right) = O(|\mathcal{T}| \cdot \alpha^K \cdot \log n_K) \tag{8}$$

(assuming that $|\mathcal{T}_1^{K-1}| \ll |\mathcal{T}|$). Thus, finally, we achieve a smaller overhead with respect to ordinary multiplication, $\log n_K$, regardless of the type of sparsity.

Act-RwCl$^+$ in Fig. 9 is the simplified vector-matrix multiplication algorithm for matrices arising from a Kronecker sum. Also in this case the complexity is dominated by the innermost for-loop, where the $O(\log n_K)$ search to compute $J_K \leftarrow \Psi_K(J_{K-1}, i_K)$ is performed $\eta[\mathbf{A}_{\mathcal{T},\mathcal{T}}] = O(|\mathcal{T}| \cdot \alpha)$ times. The complexity of *Act-RwCl*$^+$ is then

$$O(|\mathcal{T}| \cdot \alpha \cdot \log n_K)$$

regardless of k, and the resulting complexity of computing $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{x} \cdot \left(\bigoplus_{k=1}^{K} \mathbf{A}^k\right)_{\mathcal{T},\mathcal{T}}$ using *Act-RwCl*$^+$ is

$$O\left(K \cdot |\mathcal{T}| \cdot \alpha \cdot \log n_K\right),$$

only a $\log n_K$ overhead with respect to ordinary multiplication.

We can define analogous algorithms *Act-ClRw* and *Act-ClRw*$^+$, with the same complexity of *Act-RwCl* and *Act-RwCl*$^+$, although spurious entries are still a disadvantage. Unfortunately, though, *Act-ClRw*, unlike *Act-Cl*$_1$ and *Act-Cl*$_2$, does not compute the entries of $\mathbf{y}$ in order. This prevents its use in a Gauss-Seidel type iteration.

5 Comparing the multiplication algorithms

Let's consider first methods based on the potential state space. Fig. 10 compares the theoretical complexities of *Pot-Sh*, *Pot-Rw*, *Pot-Cl*, *Pot-RwCl*, and ordinary multiplication, assuming $\hat{\mathcal{T}} = \mathcal{T}$ and $K = 4$. While it is clear that *Pot-Sh* is much superior for large values of α, *Pot-Sh*, *Pot-Rw*, and *Pot-Cl* have the same performance, K times worse than ordinary multiplication, in the region

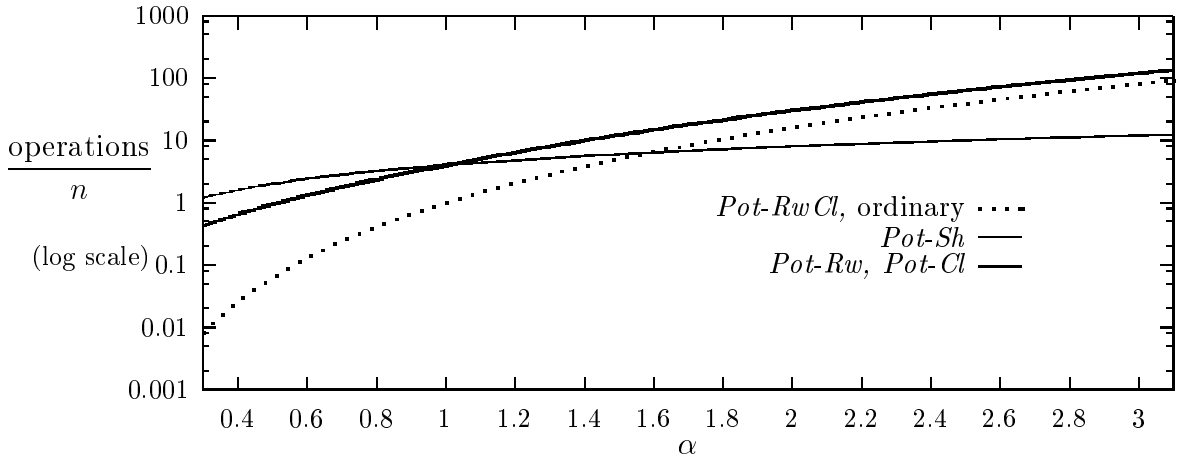

Figure 10: Comparing the complexity the multiplication algorithms ($K = 4$).

around $\alpha = 1$. Indeed, *Pot-Rw* and *Pot-Cl* do better than *Pot-Sh* when $\alpha < 1$, since they may recognize that an entire row or column of $\mathbf{A}$ is zero before reaching the innermost for-loop. On the other hand, *Pot-Sh* has the advantage to skip identity matrices completely.

Pot-Sh$^+$, *Pot-Rw*$^+$, *Pot-Cl*$^+$, and *Pot-RwCl*$^+$ have exactly the same complexity as ordinary multiplication. In other words, when computing $\mathbf{x} \cdot \mathbf{A}$ where $\mathbf{A}$ is the Kronecker product of K matrices, all of them equal to an identity matrix except one, exploiting the Kronecker structure of $\mathbf{A}$ does not result in additional overhead (since the generic entry $\mathbf{A}_{i,j}$ of $\mathbf{A}$ is obtained without performing any multiplication), nor in better efficiency (since no partial result can be reused the way *Pot-Sh* does). The same holds for the complexity of computing $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}} \cdot \bigoplus_{k=1}^{K} \mathbf{A}^k$, except that ordinary multiplication is faster if there are many merged diagonal entries in $\bigoplus_{k=1}^{K} \mathbf{A}^k$. This does not happen in our application, since, the matrices $\mathbf{R}^k$ have zero diagonals. Thus, we can conclude that using a Kronecker-based approach for the portion $\bigoplus_{k=1}^{K} \mathbf{R}^k$ of the transition rate matrix is **always a good idea**, as it does not imply additional overhead and it results in large memory savings (remember that the matrices $\mathbf{R}^k$ are usually sparse but not ultrasparse, hence $\eta[\bigoplus_{k=1}^{K} \mathbf{R}^k] \gg \sum_{k=1}^{K} \eta[\mathbf{R}^k]$).

These observations are confirmed by our running example. Table 4 gives the number of floating point multiplications performed by the algorithms we introduced and their execution times to compute $\hat{\mathbf{x}} \cdot \hat{\mathbf{R}}$, or $\hat{\mathbf{x}}_{\mathcal{T}} \cdot \hat{\mathbf{R}}_{\mathcal{T},\mathcal{T}}$, where $\hat{\mathbf{R}}$ is given by Eq. (3). Columns labeled (a) consider the decomposition into two components, where $\hat{\mathcal{T}} = \mathcal{T}$ and $\hat{\mathbf{R}}$ consists of three Kronecker products and one Kronecker sum, while columns labeled (b) refer to the second decomposition into four components, where $|\hat{\mathcal{T}}| \gg |\mathcal{T}|$ and $\hat{\mathbf{R}}$ consists of seven Kronecker products and one Kronecker sum. The Kronecker sum in (a) contains more local events and more nonzero entries than the one in (b). We list both CPU and elapsed (wall) time in seconds, for a Sun Ultra Sparc IIi under Solaris 5.6 with a 300 MHz CPU, 192 MB main memory, 4 GB disc and 4 GB swap space. A "—" in the table means that the memory requirements of the algorithm cause a thrashing effect on the machine.

Procedure	$\eta[\mathbf{R}] = 13,439,073$					
	(a) $\eta[\hat{\mathbf{R}}] = 13,439,073$			**(b)** $\eta[\hat{\mathbf{R}}] = 139,172,250$		
	mult	**CPU**	**wall**	**mult**	**CPU**	**wall**
Pot-Sh	17,955,654	13.0	13.1	—	—	—
Act-Sh	17,955,654	14.1	14.2	234,989,300	295.5	615.7
Pot-Rw	18,832,668	11.1	11.1	—	—	—
Pot-RwCl	15,714,589	3.7	3.7	—	—	—
Pot-Cl	18,832,668	11.6	11.8	243,941,600	195.9	831.0
Act-Rw$_1$	18,832,668	32.8	32.8	23,199,652	48.9	49.2
Act-Cl$_1$	18,832,668	33.6	33.7	23,199,652	52.6	53.2
Act-Rw$_2$	17,987,499	16.3	16.4	20,980,675	18.1	18.2
Act-Cl$_2$	17,987,499	20.6	20.6	20,980,675	28.2	28.4
Act-RwCl	15,714,589	8.7	8.7	15,960,012	7.0	7.1
Ordinary (single prec)	13,439,073	1.6	1.7	13,439,073	1.6	1.7

Table 4: Computational effort for vector-matrix multiplication (time in seconds).

In the first decomposition, the matrices $\mathbf{W}^k(e)$ are ultrasparse or hypersparse and, as predicted by our theoretical observations for $\alpha < 1$, *Pot-RwCl* is faster than *Pot-Sh*. *Pot-RwCl* and *Act-RwCl* perform the same number of multiplications, the increased computation time of *Act-RwCl* is caused by the binary search to evaluate $\Psi_k(J_{k-1}, j_k)$, which is necessary not only for matrices $\mathbf{A}^k \neq \mathbf{I}$ but also for identity matrices for which a matrix $\mathbf{A}^l \neq \mathbf{I}$ with index $l < k$ exists. All Kronecker-based algorithms are less computationally efficient than ordinary multiplication where $\mathbf{R}$ is stored in sparse format, which only requires $\eta[\mathbf{R}] = 13{,}439{,}073$ multiplications. However, ordinary multiplication requires about 120 MB for the sparse matrix in single precision or 180 MB in double precision, plus space for at least one iteration vector in case of a Gauss-Seidel iteration, so only single precision can be used given our hardware. This suggests that, in practice, the real advantage of Kronecker-based methods lies exclusively in their large memory savings. Of course, memory saving do become (wall) time savings when they reduce or avoid the need to rely on virtual memory.

In the second decomposition, matrices are such that $\eta[\mathbf{A}_{\hat{\mathcal{T}},\mathcal{T}}] = \eta[\mathbf{A}_{\mathcal{T},\mathcal{T}}]$, hence one should expect no significant difference between row and column algorithms while we measure 18.1 for *Act-Rw*$_2$ and 28.2 for *Act-Cl*$_2$. In a multiplication by columns, we consider one state at a time, and all the events that lead to it, while, in a multiplication by row, we consider one event at a time, and all the states it can affect. The latter way avoids having to switch between events, which makes the row algorithms perform better. The choice for the column implementation is driven by the fact that it is needed when implementing Gauss-Seidel, where entries of the new iterate of the stationary probability vector must be computed sequentially. Nevertheless since Gauss-Seidel allows to directly overwrite the operand vector by the results, the column variant uses less memory.

This is the reason why, for the second decomposition, *Pot-Cl* can still execute while *Pot-Sh* and *Pot-Rw* fail due to excessive memory requirements. However, *Pot-Cl* heavily relies on virtual memory, as the difference between CPU and wall times indicates. *Pot-Cl* considers 139,172,250 matrix entries in $\hat{\mathbf{R}}$, although only $\eta[\mathbf{R}] = 13{,}439{,}073$ are relevant. Since only a single vector of $|\hat{\mathcal{T}}|$ is needed for *Act-Sh* it behaves similarly to *Pot-Cl*. *Pot-Sh*, *Act-Sh*, *Pot-Rw*, and *Pot-Cl* are obviously inadequate when $|\hat{\mathcal{T}}| \gg |\mathcal{T}|$.

Only algorithms based on $\mathcal{T}$ run acceptably fast for the second decomposition since they do not rely on virtual memory. The results indicate that their overhead is effectively reduced from *Act-Rw*$_1$ to *Act-Rw*$_2$ to *Act-RwCl*, and from *Act-Cl*$_1$ to *Act-Cl*$_2$. Clearly, there is no reason ever to use *Act-Rw*$_1$, *Act-Rw*$_2$ or *Act-Cl*$_1$; we introduced them only as a stepping stone to the better algorithms.

In summary, *Act-RwCl* is a fast and robust algorithm, faster than *Pot-Rw* even when $|\mathcal{T}| = |\hat{\mathcal{T}}|$; it uses only $O(|\mathcal{T}|)$ memory, and makes full use of the multilevel data structure for the storage of $\mathcal{T}$. The difference between *Act-RwCl* and *Pot-RwCl* for the first decomposition indicates the overhead for the binary search introduced by Ψ. For multiplication by columns, instead, *Pot-Cl* is considerably faster than *Act-Cl*$_2$ when $|\mathcal{T}| = |\hat{\mathcal{T}}|$, but *Act-Cl*$_2$ is far superior when $|\hat{\mathcal{T}}| \gg |\mathcal{T}|$, and it uses the least amount of memory in all cases.

It should also be noted that *Act-RwCl* is faster with the second decomposition than with the first one, even if it performs slightly more operations. This is due to its $\log n_K$ overhead: n_K is 2394 in the first decomposition and 126 in the second. Clearly, the model decomposition can greatly affect the overall solution time; how to reach an optimal choice is a topic for future research.

We conclude this section by observing that we described our algorithms using K nested for-loops for illustration purposes only. Since K is model-dependent, a recursive implementation is required. To improve performance, we implemented this recursion iteratively with dynamically-allocated arrays of size K [3].

5.1 Functional rates

In the SANs proposed by Plateau et al., rates of a submodel can be a function of the state of other submodels [25, 28]. These are called functional rates and can be used in the definition of both local and synchronized events. Ways to handle functional rates include:

Local dependency. Functional rates that can be expressed as a product of local functions, $f(e, i_{[1,K]}) = \prod_{k=1}^{K} f_k(e, i_k)$, are naturally managed by the Kronecker representation considered so far. We can always achieve this "product form" by merging submodels into larger submodels [18], although, in the worst case, we might end up with a single submodel. Alternatively, we could simply store the entries corresponding to $f(e, i_{[1,K]})$, for each reachable state $i_{[1,K]}$ and for any event e that does not fit into the Kronecker framework, but this clearly has a negative impact on the memory requirements.

Representation by sum of Kronecker products. As shown in [27], any SAN with functional rates can be transformed into an equivalent SAN without functional rates by introducing

additional synchronizing events. However, the number of new synchronizing events might easily become excessive.

Generalized Kronecker products. In this case, matrix entries can be functions, not only constants. During the evaluation of this "generalized Kronecker product", the function arguments must be fixed before obtaining actual values. For any complexity result, this adds a function evaluation to the unit of measure. A recent paper [19] shows how functional rates can be integrated in the shuffle algorithm if the dependencies among submodels are acyclic, that is, if submodels can be reordered so that the functions used in submodel M_k depend only on the local states of submodels M_l, $l \leq k$. Then, functions in matrix $\mathbf{A}^k$ can be evaluated before performing the vector-matrix multiplication in step 13 of Fig. 2. The asymptotic complexity of *Pot-Sh* is unchanged, although overhead is introduced for submodel reordering and function evaluations. Cyclic dependencies can be treated as described in [19] at the price of additional complexity.

Functional rates can be easily integrated in the new multiplication algorithms we introduced. Each algorithm for Kronecker product contains K nested for-loop over $\mathbf{A}^k_{i_k,j_k} > 0$, for $k = 1, \ldots, K$. When $\mathbf{A}^k_{i_k,j_k}$ is a function f of the global state, the condition $\mathbf{A}^k_{i_k,j_k} > 0$ must be considered satisfied as long as there are local states $i_{k+1}, \ldots, i_K$ such that $f(i_1, \ldots, i_K) > 0$, where the local states $i_1, \ldots, i_k$ are fixed by the outer for-loops.

Furthermore, the statement $a_k \leftarrow a_{k-1} \cdot \mathbf{A}^k_{i_k,j_k}$ can only be performed if a_{k-1} and $\mathbf{A}^k_{i_k,j_k}$ are constants. If $\mathbf{A}^k_{i_k,j_k}$ is a function depending on the state of a submodel M_l, $l > k$, its evaluation must be postponed until the for-loop on index i_l is reached. In other words, the function evaluation sinks deeper into the nested for-loops. In the extreme case, arbitrary functional dependencies (including cyclic ones) can be managed, but all the functions must be evaluated in the innermost for-loop, where the assignment to a_K becomes $a_K \leftarrow \prod_{k=1}^{K} \mathbf{A}^k_{i_k,j_k}$. At this point, of course, the complexity becomes

$$O\left(K \cdot \eta[\mathbf{A}]\right),$$

provided functional rates cannot evaluate to zero. If they can, the innermost assignment will be reached many times only to find out that a_k should be assigned zero, and the only bound we can obtain is $O(K \cdot n^2)$. This points out how functional rates have the potential of greatly increasing the execution times, if not used sparingly and carefully.

To summarize, all algorithms presented can deal with acyclic functional rates without incurring additional complexity (except that of function evaluation, of course), just as *Pot-Sh*, provided the submodels order matches the one used in the search structure.

6 Model solution algorithms

We now return to the problem of solving a Markov model, that is, Eq. (1). In practice, $\mathbf{Q}$ is very large and indirect iterative numerical methods are employed for the solution. In all cases, starting from a guess $\boldsymbol{\pi}^{(0)}$, successive approximations $\boldsymbol{\pi}^{(m)}$ are computed, until convergence is reached.

In terms of the actual state space, the iterations we consider are:

- Power method: $\boldsymbol{\pi}^{(m+1)} \leftarrow \boldsymbol{\pi}^{(m)} \cdot (\mathbf{I} + \mathbf{Q} \cdot h^*)$, where h^* is a value slightly larger than the maximum expected sojourn time in any state, $\max_{0 \leq I < |\mathcal{T}|}\{\mathbf{h}_I\}$. Element-wise, the Power method corresponds to:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \boldsymbol{\pi}_J^{(m)} + \left(\sum_{0 \leq I < |\mathcal{T}|} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{Q}_{I,J} \right) \cdot h^*.$$

  The Power method is guaranteed to converge in theory, but it is often extremely slow.

- Jacobi method: $\boldsymbol{\pi}^{(m+1)} \leftarrow \boldsymbol{\pi}^{(m)} \cdot \mathbf{R} \cdot diag(\mathbf{h})$. Element-wise, the Jacobi method corresponds to:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \left(\sum_{0 \leq I < |\mathcal{T}|, I \neq J} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{R}_{I,J} \right) \cdot \mathbf{h}_J.$$

  The Jacobi method does not have guaranteed convergence, but it is usually faster than the Power method in practice.

- Gauss-Seidel method: $\boldsymbol{\pi}^{(m+1)} \leftarrow \boldsymbol{\pi}^{(m)} \cdot \mathbf{L} \cdot (diag(\mathbf{h})^{-1} - \mathbf{U})^{-1}$ for forward Gauss-Seidel, or $\boldsymbol{\pi}^{(m+1)} \leftarrow \boldsymbol{\pi}^{(m)} \cdot \mathbf{U} \cdot (diag(\mathbf{h})^{-1} - \mathbf{L})^{-1}$ for backward Gauss-Seidel, where $\mathbf{L}$ and $\mathbf{U}$ are strictly lower and upper triangular matrices satisfying $\mathbf{L} + \mathbf{U} = \mathbf{R}$. Element-wise, (forward) Gauss-Seidel corresponds to:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \left(\sum_{0 \leq I < J} \boldsymbol{\pi}_I^{(m+1)} \cdot \mathbf{R}_{I,J} + \sum_{J < I < |\mathcal{T}|} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{R}_{I,J} \right) \cdot \mathbf{h}_J.$$

  Gauss-Seidel does not have guaranteed convergence either, but it is guaranteed to be faster than the Jacobi method (if they converge), so it is considered the best among these three methods. Its convergence rate, however, is affected by the order in which the states are considered.

In the sequel we consider only the basic iterative techniques Jacobi and Gauss-Seidel to compare the different algorithms for vector matrix multiplication. This is because they allow us to focus on what is, algorithmically, our main concern: whether we can update the entries of the iteration vector in arbitrary fashion (as allowed by Jacobi), or whether we must update its entries sequentially (as required by Gauss-Seidel). More advanced numerical methods are discussed in Sect. 6.8.

6.1 Using the multiplication algorithms to solve a CTMC

The first choice in a Kronecker-based solution is whether to use data structures of size $|\hat{\mathcal{T}}|$ or $|\mathcal{T}|$. Initial efforts have adopted the former approach [16, 17, 26], using a probability vector $\hat{\boldsymbol{\pi}} \in \mathbb{R}^{|\hat{\mathcal{T}}|}$ initialized so that only states in $\mathcal{T}$ have nonzero probability (e.g., the initial state has probability

one). This is required because, even if we assume that the CTMC is ergodic, that is, $\mathcal{T}$ is a single recurrent class, $\hat{\mathcal{T}}$ might instead contain multiple recurrent classes. By ensuring that all the probability mass is in the class corresponding to $\mathcal{T}$ at the beginning of the iterations, we guarantee that this is true upon convergence as well. Entries $\hat{\boldsymbol{\pi}}_i = 0$ correspond to unreachable states and have no effect on the solution.

Power or Jacobi methods allow accessing the matrix $\mathbf{R}$ in an arbitrary fashion. Since they compute the entries of a new iterate $\boldsymbol{\pi}^{(m+1)}$ incrementally, using the values of the previous iterate $\boldsymbol{\pi}^{(m)}$, as pointed out in Sect. 3, double-precision vectors should be used.

The use of Gauss-Seidel requires instead computing $\boldsymbol{\pi}^{(m+1)}_{\{1,\ldots I-1\}}$ before $\boldsymbol{\pi}^{(m+1)}_I$, with the advantage that single-precision vectors can be used. This can be accomplished if we have access to $\mathbf{R}$ by columns, so only *Pot-Cl*, *Act-Cl*$_1$, and *Act-Cl*$_2$ can be used with Gauss-Seidel.

We now examine the timing requirements of the various solution algorithms, according to:

- Whether they are based on the potential, *Pot-*, or actual, *Act-*, state space.
- The type of multiplication algorithm they use.
- The type of iteration method, Jacobi (*JCB*) or Gauss-Seidel (*GSD*) they implement.

We indicate the resulting algorithms as *Pot-Sh-JCB* [5, 19], *Act-Sh-JCB*, *Pot-RwCl-JCB*, *Act-RwCl-JCB*, *Pot-Cl-GSD*, and *Act-Cl*$_2$*-GSD*. In the original SAN paper [25] introducing the Kronecker-based solution approach, the Power method is used instead of Jacobi. Thus, we present first the Jacobi method using function *Pot-Sh* to realize the iteration. Fig. 11 and 12 list only the statements within the main iteration of the numerical method, that is, the computation of a new iterate given the current one.

We also compare the space used by the various algorithms, ignoring the memory needed to store the matrices $\mathbf{R}^k$ and $\mathbf{W}^k(e)$, which are necessary in all the algorithms we consider, and are in any case negligible compared to the storage for the required vectors. For simplicity, we assume that $rate(e)$ for a synchronizing event e is equal one, and ignore it from now on; if its value were not one, we could simply incorporate it into exactly one of the matrices $\mathbf{W}^k(e)$, different from $\mathbf{I}$, for some k.

An alternative to avoid storing $\mathbf{R}$ explicitly is simply to generate it "on-the-fly" at every iteration, directly from the high-model specification. While a Jacobi-style iteration is most natural, Deavours and Sanders [15] have shown how to use a variant of Gauss-Seidel in conjunction with an on-the-fly approach for a set of modeling formalisms including GSPNs, stochastic activity networks, and stochastic reward nets. A similar idea is also in [24]. However, we do not consider this approach further since its time complexity is at least as high as that of the algorithms we present and events requiring no time (e.g., immediate transitions in GSPNs [1, 2]) cause additional overhead, by requiring the exploration of a path, not a single event, to generate a single entry.

Pot-Sh-JCB(in: $\hat{\boldsymbol{\pi}}^{old}, n_{[1,K]}, \mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \hat{\mathbf{h}}$; out: $\hat{\boldsymbol{\pi}}^{new}$)

1. $\hat{\boldsymbol{\pi}}^{new} \leftarrow \mathbf{0}$;
2. for each $e \in \mathcal{E}_S$
3. $\quad \hat{\boldsymbol{\pi}}^{aux} \leftarrow \hat{\boldsymbol{\pi}}^{old}$;
4. $\quad$ *Pot-Sh*$(n_{[1,K]}, \mathbf{W}^{[1,K]}(e), \hat{\boldsymbol{\pi}}^{aux}, \hat{\boldsymbol{\pi}}^{aux})$;
5. $\quad \hat{\boldsymbol{\pi}}^{new} \leftarrow \hat{\boldsymbol{\pi}}^{new} + \hat{\boldsymbol{\pi}}^{aux}$;
6. for $k = 1$ to K
7. $\quad$ *Pot-Sh*$^+(\hat{\boldsymbol{\pi}}^{old}, n_1^{k-1}, n_k, n_{k+1}^K, \mathbf{R}^k, \hat{\boldsymbol{\pi}}^{new})$;
8. for $i = 0$ to $n_1^K - 1$
9. $\quad$ if $\hat{\boldsymbol{\pi}}_i^{new} > 0$ then
10. $\quad\quad \hat{\boldsymbol{\pi}}_i^{new} \leftarrow \hat{\boldsymbol{\pi}}_i^{new} \cdot \hat{\mathbf{h}}_i$;

Act-Sh-JCB(in: $\boldsymbol{\pi}^{old}, n_{[1,K]}, \mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \mathcal{T}, \mathbf{h}$; out: $\boldsymbol{\pi}^{new}$)

1. $\boldsymbol{\pi}^{new} \leftarrow \mathbf{0}$;
2. for each $e \in \mathcal{E}_S$
3. $\quad \hat{\boldsymbol{\pi}}^{aux} \leftarrow map(\boldsymbol{\pi}^{old}, \mathcal{T})$;
4. $\quad$ *Pot-Sh*$(n_{[1,K]}, \mathbf{W}^{[1,K]}(e), \hat{\boldsymbol{\pi}}^{aux}, \hat{\boldsymbol{\pi}}^{aux})$;
5. $\quad \boldsymbol{\pi}^{new} \leftarrow \boldsymbol{\pi}^{new} + map^{-1}(\hat{\boldsymbol{\pi}}^{aux}, \mathcal{T})$;
6. for $k = 1$ to K
7. $\quad \hat{\boldsymbol{\pi}}^{aux} \leftarrow map(\boldsymbol{\pi}^{old}, \mathcal{T})$;
8. $\quad$ *Pot-Sh*$^+(\hat{\boldsymbol{\pi}}^{aux}, n_1^{k-1}, n_k, n_{k+1}^K, \mathbf{R}^k, \hat{\boldsymbol{\pi}}^{aux})$;
9. $\quad \boldsymbol{\pi}^{new} \leftarrow \boldsymbol{\pi}^{new} + map^{-1}(\hat{\boldsymbol{\pi}}^{aux}, \mathcal{T})$;
10. for $i = 0$ to $|\mathcal{T}| - 1$
11. $\quad$ if $\boldsymbol{\pi}_i^{new} > 0$ then
12. $\quad\quad \boldsymbol{\pi}_i^{new} \leftarrow \boldsymbol{\pi}_i^{new} \cdot \mathbf{h}_i$;

Figure 11: Algorithms *Pot-Sh-JCB* and *Act-Sh-JCB*.

6.2 Algorithms *Pot-Sh-JCB* and *Act-Sh-JCB*

The shuffle based algorithms in Fig. 11 implement the Jacobi method using *Pot-Sh* and *Pot-Sh*$^+$ for the vector-matrix multiplications. The difference is that for *Act-Sh-JCB* only the auxiliary vector has dimension $\hat{\mathcal{T}}$, hence the function *map* maps entries from the vectors $\boldsymbol{\pi}^{old}$ and $\boldsymbol{\pi}^{new}$, of dimension $|\mathcal{T}|$, to those of $\hat{\boldsymbol{\pi}}^{aux}$, of dimension $|\hat{\mathcal{T}}|$, according to Ψ. *Pot-Sh-JCB* can be considered as a special case where $\mathcal{T} = \hat{\mathcal{T}}$ and *map* is the identity. The time complexity of one *Pot-Sh-JCB* iteration is independent of the submodel ordering, and it has been obtained from the equations in Sect. 3 by substituting α with the specific number of nonzeros for each matrix involved in the computation:

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \left(\eta[\mathbf{R}^k] + \sum_{e:\mathbf{W}^k(e)\neq\mathbf{I}} \eta[\mathbf{W}^k(e)]\right)\right). \tag{9}$$

For *Act-Sh-JCB*, we need to account for the overhead due to the mapping, which is required only in vector assignment operations, where the entire set $\mathcal{T}$ is accessed sequentially. If the mapping is represented by a data structure that can enumerate the indices of the reachable states in $\hat{\mathcal{T}}$ in constant time, the overhead caused by the mapping is only a constant factor.

Even for map^{-1} one need not scan $\hat{\boldsymbol{\pi}}^{aux}$, as it is sufficient to enumerate positions in $\boldsymbol{\pi}^{new}$ sequentially and collect the corresponding values from $\hat{\boldsymbol{\pi}}^{aux}$, so that no search is required. Consequently the time complexity of one *Act-Sh-JCB* equals the complexity of *Pot-Sh-JCB*.

Memory-wise, *Pot-Sh-JCB* requires four vectors of length n: one for the expected holding times, $\hat{\mathbf{h}}$, one each for the previous and the new iteration vectors, $\hat{\boldsymbol{\pi}}^{old}$ and $\hat{\boldsymbol{\pi}}^{new}$, plus one auxiliary vector used when calling procedure *Pot-Sh*, $\hat{\boldsymbol{\pi}}^{aux}$. As already noted in Sect. 3.1, the shuffle permutation ensures that *Pot-Sh* can be called using a single vector for both input and output. Additionally,

Pot-RwCl-JCB(in: $\hat{\boldsymbol{\pi}}^{old}, n_{[1,K]}, \mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \hat{\mathbf{h}}$; out: $\hat{\boldsymbol{\pi}}^{new}$)

1. $\hat{\boldsymbol{\pi}}^{new} \leftarrow \mathbf{0}$;
2. for each $e \in \mathcal{E}_S$
3. $\quad Pot\text{-}RwCl(\hat{\boldsymbol{\pi}}^{old}, n_{[1,K]}, \mathbf{W}^{[1,K]}(e), \hat{\boldsymbol{\pi}}^{new})$;
4. for $k = 1$ to K
5. $\quad Pot\text{-}RwCl^{+}(\hat{\boldsymbol{\pi}}^{old}, n_1^{k-1}, n_k, n_{k+1}^{K}, \mathbf{R}^k, \hat{\boldsymbol{\pi}}^{new})$;
6. for $i \equiv i_{[1,K]} = 0$ to $n_1^K - 1$
7. $\quad$ if $\hat{\boldsymbol{\pi}}_i^{new} > 0$ then
8. $\quad\quad \hat{\boldsymbol{\pi}}_i^{new} \leftarrow \hat{\boldsymbol{\pi}}_i^{new} \cdot \hat{\mathbf{h}}_i$;

Act-RwCl-JCB(in: $\boldsymbol{\pi}^{old}, \mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \mathcal{T}, \mathbf{h}$; out: $\boldsymbol{\pi}^{new}$)

1. $\boldsymbol{\pi}^{new} \leftarrow \mathbf{0}$;
2. for each $e \in \mathcal{E}_S$
3. $\quad Act\text{-}RwCl(\boldsymbol{\pi}^{old}, \mathbf{W}^{[1,K]}(e), \mathcal{T}, \boldsymbol{\pi}^{new})$;
4. for $k = 1$ to K
5. $\quad Act\text{-}RwCl^{+}(\boldsymbol{\pi}^{old}, k, \mathbf{R}^k, \mathcal{T}, \boldsymbol{\pi}^{new})$;
6. for $I = 0$ to $|\mathcal{T}| - 1$
7. $\quad \boldsymbol{\pi}_I^{new} \leftarrow \boldsymbol{\pi}_I^{new} \cdot \mathbf{h}_I$;

Pot-Cl-GSD(in: $n_{[1,K]}, \mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \hat{\mathbf{h}}$; inout: $\hat{\boldsymbol{\pi}}$)

1. for $j \equiv j_{[1,K]} = 0$ to $n_1^K - 1$
2. $\quad \hat{\boldsymbol{\pi}}_j \leftarrow 0$;
3. for each $e \in \mathcal{E}_S$
4. $\quad Pot\text{-}ClEl(j_{[1,K]}, \hat{\boldsymbol{\pi}}, n_{[1,K]}, \mathbf{W}^{[1,K]}(e), \hat{\boldsymbol{\pi}}_j)$;
5. for $k = 1$ to K
6. $\quad Pot\text{-}ClEl^{+}(n_k, n_{k+1}^K, \sum_{m=1}^{k-1} j_m \cdot n_{m+1}^{k-1}, j_k, \sum_{m=k+1}^{K} j_m \cdot n_{m+1}^K, \hat{\boldsymbol{\pi}}, \mathbf{R}^k, \hat{\boldsymbol{\pi}}_j)$;
7. if $\hat{\boldsymbol{\pi}}_j > 0$ then
8. $\quad \hat{\boldsymbol{\pi}}_j \leftarrow \hat{\boldsymbol{\pi}}_j \cdot \hat{\mathbf{h}}_j$;

Act-Cl$_2$*-GSD*(in: $\mathbf{R}^{[1,K]}, \mathbf{W}^{[1,K]}(\mathcal{E}), \mathcal{T}, \mathbf{h}$; inout: $\boldsymbol{\pi}$)

1. for each $j_{[1,K]} \in \mathcal{T}$
2. $\quad J \leftarrow \Psi(j_{[1,K]})$;
3. $\quad \boldsymbol{\pi}_J \leftarrow 0$;
4. for each $e \in \mathcal{E}_S$
5. $\quad Act\text{-}ClEl_2(j_{[1,K]}, \boldsymbol{\pi}, \mathbf{W}^{[1,K]}(e), \boldsymbol{\pi}_J)$;
6. for $k = 1$ to K
7. $\quad Act\text{-}ClEl_2^{+}(j_{[1,K]}, \boldsymbol{\pi}, k, \mathbf{R}^k, \boldsymbol{\pi}_J)$;
8. $\quad \boldsymbol{\pi}_J \leftarrow \boldsymbol{\pi}_J \cdot \mathbf{h}_J$;

Figure 12: Algorithms *Pot-RwCl-JCB*, *Act-RwCl-JCB*, *Pot-Cl-GSD*, and *Act-Cl*$_2$*-GSD*.

two vector $\mathbf{z}$ and $\mathbf{z}'$ are needed in the procedures *Pot-Sh* and *Pot-Sh*$^+$, but they are only of size $\max_{1\le k\le K}(n_k)$, much smaller than n. Of these, $\hat{\boldsymbol{\pi}}^{new}$, $\hat{\boldsymbol{\pi}}^{aux}$, and $\mathbf{z}'$ should be stored in double-precision, because they are used as accumulators. For *Act-Sh-JCB*, space for vectors $\mathbf{h}$, $\boldsymbol{\pi}^{old}$, and $\boldsymbol{\pi}^{new}$ is reduced to $|\mathcal{T}|$. Additionally, an appropriate representation of $\mathcal{T}$ is required to ensure fast sequential access, e.g., an integer vector of length $\mathcal{T}$.

6.3 Algorithm *Pot-RwCl-JCB*

Pot-RwCl-JCB is the simplest iteration, it uses the *Pot-RwCl* and *Pot-RwCl*$^+$ vector-matrix multiplications by squares presented in Fig. 5 and, just as algorithm *Pot-Sh-JCB*, it uses vectors of length n. Its complexity depends on the order of the components:

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \eta[\mathbf{R}^k] + \sum_{e\in\mathcal{E}_S}\sum_{k=1}^{K}\prod_{h=1}^{k}\eta\left[\mathbf{W}^h(e)\right]\right).$$

Pot-RwCl-JCB requires three vectors vectors of size n: $\hat{\boldsymbol{\pi}}^{old}$, $\hat{\boldsymbol{\pi}}^{new}$, and $\hat{\mathbf{h}}$; only $\hat{\boldsymbol{\pi}}^{new}$ is used to accumulate sums.

6.4 Algorithm *Act-RwCl-JCB*

Act-RwCl-JCB has the same convergence behavior of *Pot-Sh-JCB*, *Act-Sh-JCB*, and *Pot-RwCl-JCB*, but uses data structures of size $|\mathcal{T}|$ by employing the *Act-RwCl* and *Act-RwCl*$^+$ vector-matrix multiplications presented in Fig. 9. The complexity of one *Act-RwCl-JCB* iteration is

$$O\left(\left(\eta\left[\left(\bigoplus_{k=1}^{K}\mathbf{R}^k\right)_{\mathcal{T},\mathcal{T}}\right] + \sum_{e\in\mathcal{E}_S}\eta\left[\left(\bigotimes_{k=1}^{K}\mathbf{W}^k(e)\right)_{\mathcal{T},\mathcal{T}}\right]\right)\cdot\log n_K\right).$$

If the number of merged entries in the above expression is negligible, this simplifies to

$$O(\eta[\mathbf{R}]\cdot\log n_K),$$

that is, just a $\log n_K$ factor over the complexity of a traditional Jacobi iteration where $\mathbf{R}$ is stored explicitly. The memory requirements of *Act-RwCl-JCB* are the same as for *Pot-RwCl-JCB*, except that vectors are now of size $|\mathcal{T}|$, not n.

6.5 Algorithm *Pot-Cl-GSD*

With the Gauss-Seidel method, the old and the new iterate can be stored into the same vector. If $\mathbf{R}$ were described by a single Kronecker product $\bigotimes_{k=1}^{K}\mathbf{A}^k$, *Pot-Cl-GSD* would be achieved by the simple call *Pot-Cl*$(\hat{\boldsymbol{\pi}}, n_{[1,K]}, \mathbf{A}^{[1,K],l}, \hat{\boldsymbol{\pi}})$, followed by the same elementwise multiplication of $\hat{\boldsymbol{\pi}}$ by the expected holding times, as performed by *Pot-RwCl-JCB*. However, $\mathbf{R}$ consists of the sum of several Kronecker products, which can be processed sequentially in a Jacobi iteration, but not in a Gauss-Seidel iteration, since we must now complete the computation of $\hat{\boldsymbol{\pi}}_i$ before starting that

Procedure	iteration vectors	holding time vector	auxiliary vectors	search data structure
Pot-Sh-JCB	$n\cdot$ (S+D)	$n\cdot$ S	$n\cdot$ (D)	—
Act-Sh-JCB	$\lvert\mathcal{T}\rvert\cdot$ (S+D)	$\lvert\mathcal{T}\rvert\cdot$ S	$n\cdot$ (D)	$\approx \lvert\mathcal{T}\rvert\cdot$ L
Pot-RwCl-JCB	$n\cdot$ (S+D)	$n\cdot$ S	—	—
Act-RwCl-JCB	$\lvert\mathcal{T}\rvert\cdot$ (S+D)	$\lvert\mathcal{T}\rvert\cdot$ S	—	$\approx \lvert\mathcal{T}\rvert\cdot$ L
Pot-Cl-GSD	$n\cdot$ S	$n\cdot$ S	—	—
Act-Cl$_2$-GSD	$\lvert\mathcal{T}\rvert\cdot$ S	$\lvert\mathcal{T}\rvert\cdot$ S	—	$\approx \lvert\mathcal{T}\rvert\cdot$ L

Table 5: Memory requirements for model solution algorithms.

of $\hat{\boldsymbol{\pi}}_{i+1}$. Hence, *Pot-Cl-GSD* must call the functions *Pot-ClEl* and *Pot-ClEl*$^+$ directly, not through *Pot-Cl* or *Pot-Cl*$^+$.

The complexity of *Pot-Cl-GSD* exceeds that of *Pot-RwCl-JCB* since it forces us to use *Pot-Cl* and not the more efficient interleaving of rows and columns *Pot-RwCl*.

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \eta[\mathbf{R}^k] + \sum_{e\in\mathcal{E}_S} \sum_{j=1}^{n} \sum_{k=1}^{K} \prod_{h=1}^{k} \eta\left[\mathbf{W}^h_{\hat{\mathcal{T}}^h,j}(e)\right]\right)$$

This indicates a tradeoff between a faster multiplication algorithm for Jacobi and a possibly reduced number iteration steps due to better convergence for Gauss-Seidel. Gauss-Seidel also requires less space, since only one vector, $\hat{\boldsymbol{\pi}}$, is required in addition to the expected holding times $\hat{\mathbf{h}}$. Furthermore, we can store $\hat{\boldsymbol{\pi}}$ in single-precision.

6.6 Algorithm *Act-Cl$_2$-GSD*

The comments made for *Pot-Cl-GSD* apply to *Act-Cl$_2$-GSD* as well. As observed before, the interleaving or rows and columns cannot be used, so we are forced to use *Act-Cl$_2$* and *Act-Cl$_2^+$*, whose amortization of the logarithmic search is less effective. This points out a surprising tradeoff between *Act-RwCl-JCB*, which has slower convergence but a smaller overhead, $\log n_K$, and *Act-Cl$_2$-GSD*, which has better numerical convergence but higher overhead, possibly as high as $\log n$.

The complexity of *Act-Cl$_2$-GSD* is

$$O\left(|\mathcal{T}| \cdot \sum_{k=1}^{K} \left(\frac{\eta\left[\left(\mathbf{I}_{n_1^{k-1}} \otimes \mathbf{R}^k\right)_{\mathcal{T}_1^{k-1}\times\mathcal{T}^k,\mathcal{T}_1^k}\right] \cdot \sum_{h=k}^{K} \log n_h + \sum_{e\in\mathcal{E}_S} \eta\left[\left(\bigotimes_{h=1}^{k} \mathbf{W}^h(e)\right)_{\mathcal{T}_1^{k-1}\times\mathcal{T}^k,\mathcal{T}_1^k}\right] \cdot \log n_k}{|\mathcal{T}_1^k|} \right)\right).$$

6.7 Comparing the model solution algorithms

Table 5 summarizes the memory requirements for the solution algorithms we considered, expressed in the units S and D, for a single- or double-precision floating point number (usually 4 and 8 bytes,

Procedure	(a) vectors	(a) extra	(b) vectors	(b) extra
Pot-Sh-JCB	24,974,208	12,487,104	276,595,200	138,297,600
Act-Sh-JCB	24,974,208	18,730,656	24,974,208	144,541,100
Pot-RwCl-JCB	24,974,208	—	276,595,200	—
Act-RwCl-JCB	24,974,208	3,127,000	24,974,208	3,804,700
Pot-Cl-GSD	12,487,104	—	138,297,600	—
Act-Cl_2-GSD	12,487,104	3,127,000	12,487,104	3,804,700
Ordinary-GSD	12,487,104	126,243,240	12,487,104	126,243,240

Table 6: Memory requirements (in bytes) for our example.

Procedure	(a) CPU	(a) wall	(a) iter	(b) CPU	(b) wall	(b) iter
Pot-Sh-JCB	5,102	5,124	387	—	—	—
Act-Sh-JCB	5,481	5,490	387	114,358	238,275	387
Pot-RwCl-JCB	1,431	1,434	387	—	—	—
Act-RwCl-JCB	3,360	3,366	387	2,741	2,757	387
Pot-Cl-GSD	2,200	2,232	189	28,407	120,483	144
Act-Cl_2-GSD	3,900	3,907	189	4,132	4,157	144
Ordinary-GSD	310	325	189	237	247	144

Table 7: Execution times (seconds) and number of iterations for our example.

respectively), and L, for a local state of M_K (usually 1 or 2 bytes). The actual memory usage for our running example is instead in Table 6, for decompositions (a) and (b). Column "vectors" lists the memory (in bytes) for the iteration vectors and $\mathbf{h}$; column "extra" for auxiliary vectors or search data structures.

The timing results are in Table 7. We performed iterations using the absolute convergence criterion $||\boldsymbol{\pi}^{old} - \boldsymbol{\pi}^{new}||_\infty < 10^{-8}$.

As already anticipated in Table 4, algorithms *Pot-Sh-JCB* and *Pot-RwCl-JCB* fail due to insufficient memory with the second decomposition, while *Pot-Cl-GSD* could be run, but with an unacceptable amount of overhead; the same holds for *Act-Sh-JCB*, where the space requirements are clearly dominated by the auxiliary vector $\hat{\boldsymbol{\pi}}^{aux}$ of size n.

We observe that the two decompositions result in different state orderings, which in turn affect the convergence of *Act-Cl_2-GSD*. Hence, 189 iterations are required for the first decomposition, but only 144 for the second one.

6.8 Advanced numerical methods

Relaxation can be applied to the Jacobi and Gauss-Seidel methods to improve convergence, yielding the JOR and SOR methods. The new iteration vector in step $m+1$ is then computed as

$(1-\omega)\boldsymbol{\pi}^{(m)} + \omega\boldsymbol{\pi}^{(m+1)}$, where $\boldsymbol{\pi}^{(m+1)}$ results from a Jacobi or Gauss-Seidel iteration step and $0 < \omega < 2$. Apart from the above classical stationary techniques, "projection techniques" have recently become very popular for the analysis of CTMCs [28]. These approximate the exact solution using a sequence of vectors on a subspace with a smaller dimension (often the so-called Krylov subspace $\mathcal{L} = \text{span}\{\boldsymbol{\pi} \cdot \mathbf{Q}, \ldots, \boldsymbol{\pi} \cdot \mathbf{Q}^m\}$ is used). Different projection methods have been proposed in the literature for the solution of CTMCs, among them are the generalized minimal residual method (GMRES), the conjugate gradient squared method (CGS), the biconjugate gradient stabilized method (BiCGSTAB) and the quasi minimal residual method (QMR), see [28]. Since their main computational effort remains the vector-matrix multiplications, our algorithms can still be applied if the matrix is represented as a sum of tensor products of submodel matrices.

However, although projection methods often exhibit faster convergence, they do have drawbacks, especially when applied in the context of Kronecker-based CTMC analysis. First, they require to store additional vectors. GMRES's memory requirements are so massive that it cannot be used for really large CTMCs, but even the other methods require at least five additional vectors. This essentially doubles the memory requirements (see Tab. 5 and 6). A second problem when using projection methods is the requirement for preconditioners. Usually, projection methods are applied to the matrix $\mathbf{M}^{-1} \cdot \mathbf{Q}$ or $\mathbf{Q} \cdot \mathbf{M}^{-1}$, where $\mathbf{M}^{-1}$ is some easy-to-compute approximation of the generalized inverse of $\mathbf{Q}$ [28]. Standard preconditioners based on an incomplete LU factorization of $\mathbf{Q}$ cannot be used in our case, as they would destroy the Kronecker representation of $\mathbf{Q}$. Preliminary results on preconditioners that can be represented as sum of Kronecker products exist [8, 28] but, as stated on p. 490 of [28]: "Much more research needs to be conducted into finding preconditioning techniques that can be applied to SAN descriptors without the need to expand the global generator". Our own experience with the projection techniques mentioned indicates that, without preconditioning, they do not outperform JOR or SOR [8].

Apart from projection techniques, aggregation/disaggregation is a promising way of speeding up the convergence of iterative techniques. An approach in the context of Kronecker-based analysis is proposed in [6]. As the most time-consuming step is still the computation of vector-matrix products, the algorithms presented in this paper can be applied. An alternative approach to realize Gauss-Seidel and block Gauss-Seidel for SANs has been recently proposed [30], where the matrices $\mathbf{L}$ and $\mathbf{U}$ are each represented as sums of Kronecker products. Our algorithms can be applied also to this approach, since the problem of unreachable states in $\hat{\mathcal{T}}$ is still present and, due to the LU splitting, the matrices in the Kronecker products are even more sparse than usual.

7 Conclusion

We presented a comprehensive set of Kronecker-based vector-matrix multiplication and solution algorithms for structured Markov models in a unified framework that ignores the peculiarities of specific modeling formalisms. Time and space complexities are given, with special attention to the sparsity of the involved matrices.

We have shown how the Kronecker-based solution of structured Markov models can be carried

out with smaller memory and execution complexity than previously proposed. This is achieved by exploiting the sparsity of the matrices involved in the Kronecker operations, by considering the actual state space instead of the potential state space (which can contain many unreachable states), by adopting a sophisticated data structure to determine whether a state is reachable or not, and by performing vector-matrix multiplications by rows or by columns, thus enabling the use of both Jacobi- and Gauss-Seidel-style methods.

Our results are not limited to stationary solution of ergodic models. Indeed, the computation of the cumulative sojourn time in the transient states up to absorption in an absorbing CTMC also requires the solution of (nonhomogeneous) linear system, while the iteration performed by the Uniformization method for the transient solution of a CTMC is essentially the same as that of the Power method.

The proposed algorithms have been implemented in SupGSPN, a numerical solution package contained in the APNN-toolbox [3], for the Petri net formalism, and in SMART [10], for arbitrary discrete-state formalisms. The reduced memory requirements allows us to solve very large Markov models (over 10^7 states) on a modern workstation in a matter of hours.

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comp. Syst.*, 2(2):93–122, May 1984.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets.* John Wiley & Sons, 1995.

[3] F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In R. Puigjaner, N. Savino, and B. Serra, editors, *Proc. 10th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science 1469, pages 356–359. Springer-Verlag, 1998.

[4] P. Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In G. Balbo and G. Serazzi, editors, *Computer performance evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[5] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems.*, 15:59–80, 1994.

[6] P. Buchholz. An aggregation/disaggregation algorithm for stochastic automata networks. *Probability in the Engineering and Informational Sciences*, 11(2):229–253, 1997.

[7] P. Buchholz. Hierarchical structuring of superposed GSPNs. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 81–90, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[8] P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.

[9] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. In C. Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models*, IMA Volumes in Mathematics and its Applications 48, pages 145–191. Springer-Verlag, 1993.

[10] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, page 60, Urbana-Champaign, IL, USA, Sept. 1996. IEEE Comp. Soc. Press.

[11] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science 1245, pages 44–57, St. Malo, France, June 1997. Springer-Verlag.

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

[13] M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30:116–125, Feb. 1981.

[14] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving very large Markov models. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science 1245, pages 58–71, St. Malo, France, June 1997. Springer-Verlag.

[15] D. D. Deavours and W. H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 132–141, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[16] S. Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Perf. Eval.*, 18:21–26, 1993.

[17] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets)*, Lecture Notes in Computer Science 815, pages 258–277, Zaragoza, Spain, June 1994. Springer-Verlag.

[18] P. Fernandes, B. Plateau, and W. J. Stewart. Numerical issue for stochastic automata networks. In *Proc. of the 4rd Workshop on Process Algebra and Performance Modelling (PAPM)*, Torino, Italy, July 1996. CLUT.

[19] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[20] R. A. Howard. *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes*. John Wiley and Sons, 1971.

[21] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. Softw. Eng.*, 22(4):615–628, Sept. 1996.

[22] P. Kemper. Reachability analysis based on structured representations. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996 (Proc. 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka, Japan)*, Lecture Notes in Computer Science 1091, pages 269–288. Springer-Verlag, June 1996.

[23] P. Kemper. *Superposition of generalized stochastic Petri nets and its impact on performance analysis*. PhD thesis, Universität Dortmund, 1996.

[24] B. Lubachevsky and D. Mitra. A chaotic asynchronous algorithm for computing the fixed point of nonnegative matrices with unit spectral radius. *J. ACM*, 33:130–150, 1986.

[25] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, Austin, TX, USA, May 1985.

[26] B. Plateau and K. Atif. Stochastic Automata Network for modeling parallel systems. *IEEE Trans. Softw. Eng.*, 17(10):1093–1108, Oct. 1991.

[27] B. Plateau and J.-M. Fourneau. A methodology for solving Markov models of parallel systems. *J. Par. and Distr. Comp.*, 12:370–387, 1991.

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

[29] W. J. Stewart, K. Atif, and B. Plateau. The numerical solution of stochastic automata networks. *Europ. J. of Oper. Res.*, 86:503–525, 1995.

[30] E. Uysal and T. Dayar. Iterative methods based on splittings for stochastic automata networks. *Eur. J. Op. Res.*, 110(1):166–186, 1998.