NASA/CR-97-206274
ICASE Report No. 97-66

Complexity of Kronecker Operations on Sparse Matrices with Applications to Solution of Markov Models

Peter Buchholz (Universitat Dortmund, Germany)
Gianfranco Ciardo (College of William and Mary)
Susanna Donatelli (Universita di Torino, Italy)
Peter Kemper (Universitat Dortmund, Germany)

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contracts NAS1-97046 & NAS1-19480

December 1997

COMPLEXITY OF KRONECKER OPERATIONS ON SPARSE MATRICES WITH APPLICATIONS TO THE SOLUTION OF MARKOV MODELS

PETER BUCHHOLZ[*], GIANFRANCO CIARDO[†], SUSANNA DONATELLI[‡], AND PETER KEMPER[§]

Abstract. We present a systematic discussion of algorithms to multiply a vector by a matrix expressed as the Kronecker product of sparse matrices, extending previous work in a unified notational framework. Then, we use our results to define new algorithms for the solution of large structured Markov models. In addition to a comprehensive overview of existing approaches, we give new results with respect to: (1) managing certain types of state-dependent behavior without incurring extra cost; (2) supporting both Jacobi-style and Gauss-Seidel-style methods by appropriate multiplication algorithms; (3) speeding up algorithms that consider probability vectors of size equal to the "actual" state space instead of the "potential" state space.

Key words. Kronecker algebra, Markov chains, vector-matrix multiplication

Subject classification. Computer Science

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 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 steady-state probability-vector, from which most quantities of interest can be obtained as a weighted sum by using "reward rates" as weights [17].

Although the mapping of a high-level model onto the CTMC and the computation of the steady-state 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 steady state solution of large ergodic CTMCs, that is, the computation of the vector $\boldsymbol{\pi}$, where $\boldsymbol{\pi}_i$ is the steady-state probability of state i. However, our contributions can also be used to improve the generation of the state space [8, 21] and other types of analysis such as the computation of the expected time spent in transient states up to absorption in absorbing CTMCs and transient analysis of arbitrary CTMCs [6].

$\boldsymbol{\pi} \in \mathbb{R}^{|\mathcal{T}|}$ is the solution of the system of linear equations

$$\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0} \qquad \text{subject to} \qquad \boldsymbol{\pi} \cdot \mathbf{1}^T = 1, \tag{1.1}$$

where $\mathbf{Q}$ is the generator matrix and $\mathcal{T}$ is the set of states of the CTMC.

Direct solution methods such as the well-known Gaussian elimination are not applicable, 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

[*]Informatik IV, Universität Dortmund, D-44221 Dortmund, Germany (email: *buchholz@ls4.informatik.uni-dortmund.de*).

[†]Dept. of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, USA (email: *ciardo@cs.wm.edu*). This research was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while this author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.

[‡] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy (email: *susi@di.unito.it*).

[§]Informatik IV, Universität Dortmund, D-44221 Dortmund, Germany (email: *kemper@ls4.informatik.uni-dortmund.de*).

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
$\hat{\mathbf{h}}, \mathbf{h}$	$\forall i \in \mathcal{T}\ \hat{\mathbf{h}}_i = \mathbf{h}_{\Psi(i)}$	Expected holding time vector

Table 2.1

Symbols used in the paper

is of little help, since access times to virtual memory are too long to allow an efficient implementation of iterative solution techniques (although [11] 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 sum of Kronecker products of smaller matrices that result from a high-level model structured into submodels. A general framework for this idea is described in Sect. 4. This covers several high-level formalisms to describe models in a compositional way [13, 14, 18, 24].

Solution methods exploiting a Kronecker structure are iterative and they differ from conventional iteration techniques in how they perform the required vector-matrix multiplications. Earlier approaches [24] 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. This resulted in a limited applicability, since these approaches become inefficient for models where submodel matrices are quite sparse, the actual state space is much smaller than the potential state space, or the Power method requires too many iterations.

More recent publications [18, 9] began to overcome these limitations, but there is still no systematic approach to fully exploit the potential of Kronecker-based solution techniques. In this paper, we present a family of solution techniques using 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 the Kronecker-based vector-matrix multiplication algorithms we present, both theoretically and by means of a realistic example.

In the next section, we define the notation used. Sect. 3 contains a framework for the description of Markov models and Sect. 4 considers composed Markov models and the corresponding Kronecker structure of the generator matrix. Sect. 5 presents and analyzes different algorithms to multiply a vector by a matrix represented as a Kronecker product, using a running example. Sect. 6 describes iterative solution approaches that use these multiplication algorithms to compute the steady-state solution of a CTMC.

2. Notation.

Table 2.1 summarizes the symbols used in this paper. Except for the set of real numbers, $I\!R$, 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 (for scalars) or superscripts (for sets, vectors and 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 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 $\mathbf{x}$, $\mathit{diag}(\mathbf{x})$ is a square matrix having vector $\mathbf{x}$ on the diagonal and zero elsewhere. Given an $n \times n$ matrix $\mathbf{A}$, $\mathit{rowsum}(\mathbf{A}) = \mathit{diag}(\mathbf{A} \cdot \mathbf{1}_{n\times 1})$ 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 $\mathbf{A}^k \in I\!R^{n_k \times n_k}$. Let $n_l^u = \prod_{k=l}^{u} n_k$, $n = n_1^K$, and $\bar{n}_k = n/n_k$. If we assume a mixed-base numbering scheme, 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 we assume that $i_{[1,K]}$ and $j_{[1,K]}$ are the mixed-based representation of i and j, respectively, the generic element of $\mathbf{A} \in I\!R^{n\times n}$ is

$$\mathbf{A}_{i,j} = \mathbf{A}_{i_{[1,K]},j_{[1,K]}} = \mathbf{A}^1_{i_1,j_1} \cdot \mathbf{A}^2_{i_2,j_2} \cdots \mathbf{A}^K_{i_K,j_K} \tag{2.1}$$

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 nonzero diagonal entries. 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.

3. Description and solution of a Markov model.

A formal high-level model having an underlying CTMC specifies a stochastic automaton of some sort. Instead of assuming a specific high-level formalism such as Stochastic Petri Nets [5], Queueing Networks [26], or Stochastic Process Algebra [16], we consider a generic framework where a model is a tuple:

$$M = (\hat{\mathcal{T}}, \mathcal{E}, i^0, \mathit{active}, \mathit{new}, \mathit{rate}, \mathit{weight})$$

- $\hat{\mathcal{T}}$ is the set of potential states.
- $\mathcal{E}$ is the set of possible events.
- $i^0 \in \hat{\mathcal{T}}$ is the initial state.
- $\mathit{active} : \mathcal{E} \times \hat{\mathcal{T}} \to \{True, False\}$.
- $\mathit{new} : \mathcal{E} \times \hat{\mathcal{T}} \to \hat{\mathcal{T}} \cup \{\mathsf{null}\}$; $\mathit{new}(e,i) = \mathsf{null}$ iff $\mathit{active}(e,i) = False$.
- $\mathit{rate} : \mathcal{E} \to I\!R^{+}$.

- $weight : \mathcal{E} \times \hat{\mathcal{T}} \to I\!R^+$.

If an event e is active in state i (that is, if $active(e,i) = True$) and if $new(e,i) = j$, we say that state j is reachable in one step from i. The transitive and reflexive closure of this relation is called reachability. Using the reachability relation, we can build the set $\mathcal{T} \subseteq \hat{\mathcal{T}}$ of states reachable from the initial state i^0, called the (actual) state space of M. An event e active in state i has an exponentially-distributed duration with parameter $rate(e) \cdot weight(e,i)$. Since the duration of the events is exponentially distributed, and the next state depends only on the current one (by definition of function new), M defines a CTMC.

$\mathcal{T}$ can be generated using a state-space exploration algorithm, essentially a breadth-first search of the graph implicitly defined by the model, starting from i^0; assuming that $\mathcal{T}$ is finite, the search terminates. Then, we can define a function $\Psi : \hat{\mathcal{T}} \to \{0, \ldots, |\mathcal{T}|-1, \mathsf{null}\}$, with $\Psi(i) = \mathsf{null}$ iff $i \notin \mathcal{T}$ and such that the restriction of Ψ to $\mathcal{T}$ is a bijection.

With the indirect iterative solution methods for the steady-state solution of the CTMC, it is convenient to write the infinitesimal generator $\mathbf{Q} \in I\!R^{|\mathcal{T}|\times|\mathcal{T}|}$ as

$$\mathbf{Q} = \mathbf{R} - rowsum(\mathbf{R}) = \mathbf{R} - diag(\mathbf{h})^{-1},$$

where $\mathbf{R} \in I\!R^{|\mathcal{T}|\times|\mathcal{T}|}$ is the transition rate matrix and $\mathbf{h}$ is the vector of expected holding times. $\mathbf{R}$ differs from $\mathbf{Q}$ only in its diagonal, which is zero, while $\mathbf{h}$ contains the inverse of the negative of the diagonal of $\mathbf{Q}$. $\mathbf{R}$ can be generated at the same time as $\mathcal{T}$, or in a second pass, once $|\mathcal{T}|$ is known, since we can then use an efficient sparse row-wise or column-wise format [23]; $\mathbf{h}$ is instead stored as a full vector.

In the following, we assume that $\mathbf{R}$ is irreducible, that is, the CTMC is ergodic.

4. Model composition and Kronecker description of Q. We consider structured models described as the parallel composition of a set of stochastic automaton submodels. Formally, a stochastic automaton $M = (\hat{\mathcal{T}}, \mathcal{E}, i^0, active, new, rate, weight)$ is the parallel composition of the K models $M_k = (\hat{\mathcal{T}}^k, \mathcal{E}^k, i^0_k, active^k, new^k, rate^k, weight^k)$, $1 \le k \le K$, iff:

- $\hat{\mathcal{T}} = \hat{\mathcal{T}}^1 \times \ldots \times \hat{\mathcal{T}}^K$. A state of the composed model is the tuple describing the local state of the K submodels: $i = i_{[1,K]}$.
- $\mathcal{E} = \bigcup_{k=1}^{K} \mathcal{E}^k$. Let $\mathcal{E}_S = \{e : \exists h \neq k : e \in \mathcal{E}^h \cap \mathcal{E}^k\}$ be the set of "synchronizing" events common to two or more submodels. The remaining events can be partitioned into K sets of "local events", one for each submodel: $\mathcal{E}^k_L = \mathcal{E}^k \setminus \mathcal{E}_S$.

  A submodel M_k has no influence on an event $e \notin \mathcal{E}^k$, nor is it influenced by such events. Hence, we extend the local functions $active^k$, $weight^k$, new^k, and $rate^k$ to the entire set of events $\mathcal{E}$ as follows. For any $e \notin \mathcal{E}^k$ and for any local state $i_k \in \hat{\mathcal{T}}^k$:
  - $active^k(e, i_k) = True$ (neutral element for logical conjunction).
  - $weight^k(e, i_k) = 1$ (neutral element for multiplication).
  - $new^k(e, i_k) = i_k$ (neutral operation for state changes).
  - $rate^k(e) = \infty$ (neutral element for the minimum).
- $i^0 = i^0_{[1,K]}$.
- $active(e, i_{[1,K]}) = \bigwedge_{k=1}^{K} active^k(e, i_k)$. An event local to M_k is active iff it is active in M_k; a synchronizing event e is active iff it is active in all submodels where it is defined.
- $new(e, i_{[1,K]}) = \begin{cases} \mathsf{null} & \text{if } \exists k,\ active^k(e, i_k) = False \\ j_{[1,K]} & \text{otherwise} \end{cases}$, where $j_k = new^k(e, i_k)$.

- $weight(e, i_{[1,K]}) = \prod_{k=1}^{K} weight^k(e, i_k)$.
- $rate(e) = \min_{1 \le k \le K} \{rate^k(e)\}$. The choice of using the minimum is just one possibility; for the solution algorithms we present, any function will do. For example, we could instead assign the rate on the global model, independently of the values $rate^k(e)$.

In the presence of multiple synchronizations, $\hat{\mathcal{T}}$ can be a strict superset of $\mathcal{T}$, and, in this case, some local states in $\hat{\mathcal{T}}^k$ might be unreachable. Thus, we define the "actual local" state spaces as the projection of $\mathcal{T}$ on the k-th component:

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

We can then 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 exploring $\mathcal{T}$ once.

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

- $\mathbf{W}^k(e)$ is a $n_k \times n_k$ matrix defined as

$$\mathbf{W}^k_{i_k,j_k}(e) = \begin{cases} weight^k(e, i_k) & \text{if } active^k(e, i_k) = True \text{ and } j_k = new^k(e, i_k) \\ 0 & \text{otherwise} \end{cases} .$$

- $\mathbf{R}^k$ are the local transition rate matrices describing the effect of local events:

$$\mathbf{R}^k = \sum_{e \in \mathcal{E}^k_L} rate(e) \cdot \mathbf{W}^k(e).$$

Using related frameworks, [3, 9, 14, 24, 25] 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:

$$\hat{\mathbf{R}} = \underbrace{\sum_{e \in \mathcal{E}_S} rate(e) \cdot \bigotimes_{k=1}^{K} \mathbf{W}^k(e)}_{\text{synchronizing events}} + \underbrace{\bigoplus_{k=1}^{K} \mathbf{R}^k}_{\text{local events}} . \tag{4.1}$$

The expression for $\hat{\mathbf{Q}}$ is analogous but we omit it because, in the following, we assume that only $\hat{\mathbf{R}}$ is kept in Kronecker form, and that $\mathbf{h}$, or its "potential" version $\hat{\mathbf{h}}$, is stored explicitly as a full vector. Alternatively, we could save memory by using a Kronecker description for $\mathbf{h}$, at the cost of additional execution time.

4.1. A small example. To illustrate the use of Kronecker operators for the description of $\mathbf{R}$, we consider first a simple example with $K = 2$ models M_1 and M_2. A more complex running example is introduced in the next section.

The definition of M_1 and M_2 is given in Table 4.1, the extension of functions to the entire set $\mathcal{E}$ is omitted for readability. The composed model M has a state space $\hat{\mathcal{T}} = \{(0,0),(0,1),(1,0),(1,1)\}$ and its events are $\mathcal{E} = \mathcal{E}_S \cup \mathcal{E}^1_L \cup \mathcal{E}^2_L$, where $\mathcal{E}^1_L = \{e_1\}$ is the only local event in M_1, $\mathcal{E}^2_L = \{e_2\}$ is the only local event in M_2, and $\mathcal{E}_S = \{e_3\}$ is the only synchronizing event. If we define $rate(e_1) = 6$, $rate(e_2) = 5$, $rate(e_3) = 4$, we obtain the following matrices (zero entries are omitted for readability):

model M_1:	$\mathcal{T}^1 = \{0,1\}$	$\mathcal{E}^1 = \{e_1, e_3\}$
$active^1(\cdot,\cdot) = False$ except		
$active^1(e_1,0) = True$	$new^1(e_1,0) = 1$	$weight^1(e_1,0) = 0.6$
$active^1(e_3,1) = True$	$new^1(e_3,1) = 0$	$weight^1(e_3,1) = 0.4$
model M_2:	$\mathcal{T}^2 = \{0,1\}$	$\mathcal{E}^2 = \{e_2, e_3\}$
$active^2(\cdot,\cdot) = False$ except		
$active^2(e_2,1) = True$	$new^2(e_2,1) = 0$	$weight^2(e_2,1) = 0.7$
$active^2(e_3,0) = True$	$new^2(e_3,0) = 1$	$weight^2(e_3,0) = 0.5$

TABLE 4.1
Definition of the submodels for our small example

$$\mathbf{W}^1(e_3)=\begin{array}{c|cc|}
 & 0 & 1 \\ \hline
0 & & \\
1 & 0.4 & \\ \hline
\end{array}
\qquad
\mathbf{W}^2(e_3)=\begin{array}{c|cc|}
 & 0 & 1 \\ \hline
0 & & 0.5 \\
1 & & \\ \hline
\end{array}
\qquad
\mathbf{R}^1=\begin{array}{c|cc|}
 & 0 & 1 \\ \hline
0 & & 6\cdot 0.6 \\
1 & & \\ \hline
\end{array}
\qquad
\mathbf{R}^2=\begin{array}{c|cc|}
 & 0 & 1 \\ \hline
0 & & \\
1 & 5\cdot 0.7 & \\ \hline
\end{array},$$

resulting in

$$\hat{\mathbf{R}} = 4\cdot\bigotimes_{k=1}^{2}\mathbf{W}^k(e_3) + \bigoplus_{k=1}^{2}\mathbf{R}^k = 4\cdot\big(\mathbf{W}^1(e_3)\otimes\mathbf{W}^2(e_3)\big) + \big(\mathbf{R}^1\otimes\mathbf{I}_2 + \mathbf{I}_2\otimes\mathbf{R}^2\big) =$$

$$4\cdot\begin{array}{c|cc|cc|}
 & (0,0) & (0,1) & (1,0) & (1,1) \\ \hline
(0,0) & & & & \\
(0,1) & & & & \\ \hline
(1,0) & & 0.4\cdot 0.5 & & \\
(1,1) & & & & \\ \hline
\end{array}
+\left(\begin{array}{|cc|cc|}
(0,0) & (0,1) & (1,0) & (1,1) \\ \hline
 & & 3.6 & \\
 & & & 3.6 \\ \hline
 & & & \\
 & & & \\ \hline
\end{array}
+\begin{array}{|cc|cc|}
(0,0) & (0,1) & (1,0) & (1,1) \\ \hline
 & & & \\
3.5 & & & \\ \hline
 & & & \\
 & & 3.5 & \\ \hline
\end{array}\right).$$

4.2. A running example. We now describe the running example used throughout the rest of the paper 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 being processed. Fig. 4.1 depicts it as a fork-join queuing network, with machines as queues and parts as customer classes (chains). We assume exponentially distributed service times for simplicity. c_3 can process up to three parts in parallel, c_2 up to two, and c_1 only one.

A part of type A accesses c_3 with high priority 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 with high priority 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 with high priority 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

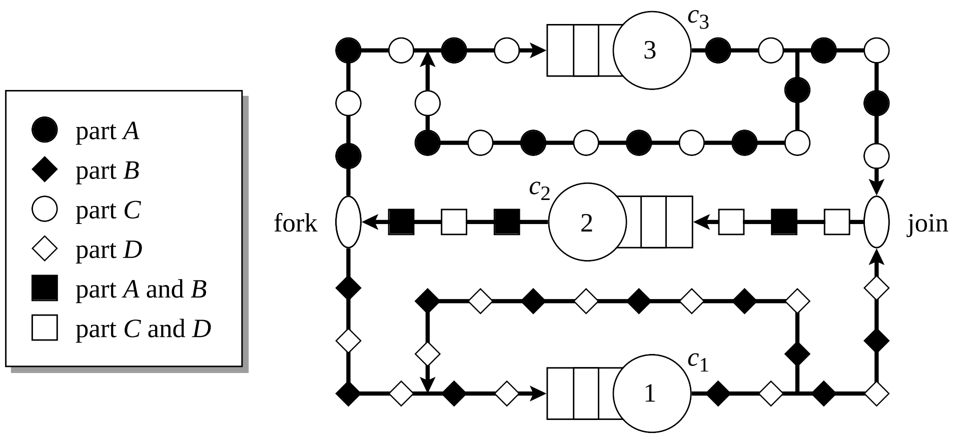


FIG. 4.1. *Multiclass Queueing Network for our running example*

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

TABLE 4.2
Number of nonzeros using the first decomposition.

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 the processing that machines c_1, c_2, and c_3 perform on the high-priority parts A and B, including their joint processing; submodel L is analogous, for the low-priority parts C and D.

The synchronizing events are $\mathcal{E}_S = \{low_1, low_2, low_3\}$, representing the low-priority parts using the three machines. 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 4.2 gives the number of nonzeros for the matrices involved in the Kronecker description of $\mathbf{R}$

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.

5. Complexity of vector-matrix multiplication. 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 $\mathbf{A}^k \in I\!R^{n_k \times n_k}, k \in \{1, \ldots, K\}$, also stored in sparse row-wise or column-wise format, as appropriate, a direct application of Eq. 2.1 requires K operations to obtain each matrix entry. If each $\mathbf{A}_{i,j}$ is computed only once for each pair (i,j) and only nonzero $\mathbf{A}_{i,j}$ elements are computed, the complexity of computing $\mathbf{x} \cdot \mathbf{A}$ becomes $O(K \cdot \eta[\mathbf{A}])$. In the next

```
Shffl(in: $n_{[1,K]}$, $\mathbf{A}^{[1,K]}$; inout: $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$);
 1. $n_{left} \leftarrow 1$;
 2. $n_{right} \leftarrow n_2^K$;
 3. for $k = 1$ to $K$
 4.   $base \leftarrow 0$;
 5.   $jump \leftarrow n_k \cdot n_{right}$;
 6.   if $\mathbf{A}^k \neq \mathbf{I}$ then
 7.     for $block = 0$ to $n_{left} - 1$
 8.       for $offset = 0$ to $n_{right} - 1$
 9.         $index \leftarrow base + offset$;
10.         for $h = 0$ to $n_k - 1$
11.           $\mathbf{z}_h \leftarrow \hat{\mathbf{x}}_{index}$;
12.           $index \leftarrow index + n_{right}$;
13.         $\mathbf{z}' \leftarrow \mathbf{z} \cdot \mathbf{A}^k$;
14.         $index \leftarrow base + offset$;
15.         for $h = 0$ to $n_k - 1$
16.           $\hat{\mathbf{y}}_{index} \leftarrow \mathbf{z}'_h$;
17.           $index \leftarrow index + n_{right}$;
18.       $base \leftarrow base + jump$;
19.     $\hat{\mathbf{x}} \leftarrow \hat{\mathbf{y}}$;
20.   $n_{left} \leftarrow n_{left} \cdot n_k$;
21.   $n_{right} \leftarrow n_{right}/n_{k+1}$;
```

```
Shffl+(in: $\mathbf{x}$, $n_{left}$, $n_k$, $n_{right}$, $\mathbf{A}^k$; inout: $\mathbf{y}$);
 1. $base \leftarrow 0$;
 2. $jump \leftarrow n_k \cdot n_{right}$;
 3. for $block = 0$ to $n_{left} - 1$
 4.   for $offset = 0$ to $n_{right} - 1$
 5.     $index \leftarrow base + offset$;
 6.     for $h = 0$ to $n_k - 1$
 7.       $\mathbf{z}_h \leftarrow \hat{\mathbf{x}}_{index}$;
 8.       $index \leftarrow index + n_{right}$;
 9.     $\mathbf{z}' \leftarrow \mathbf{z} \cdot \mathbf{A}^k$;
10.     $index \leftarrow base + offset$;
11.     for $h = 0$ to $n_k - 1$
12.       $\hat{\mathbf{y}}_{index} \leftarrow \hat{\mathbf{y}}_{index} + \mathbf{z}'_h$;
13.       $index \leftarrow index + n_{right}$;
14.   $base \leftarrow base + jump$;
```

FIG. 5.1. *Vector-matrix multiplication using perfect shuffles.*

section, we recall an algorithm that achieves a better complexity by exploiting the inherent structure of a Kronecker product.

Before doing so, however, we observe that the matrices involved in the Kronecker products of Eq. 4.1 can have very few elements. In our particular application, the matrices $\mathbf{A}^k$ are either the K matrices $\mathbf{W}^k(e)$ for a given $e \in \mathcal{E}$, or they are all the identity except for one of them being equal $\mathbf{R}^k$, for a given k. This last case arises from rewriting the Kronecker sum appearing in Eq. (4.1) into a (regular) sum of of Kronecker products, as explained in Sect. 2. 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 $\mathbf{A}^k$), where $A = \otimes A^k$:

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: $\alpha \gg 1 \Rightarrow \eta[\mathbf{A}] = n \cdot \alpha^K \gg n$ (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 $k = 1, \ldots, K$. Truly hypersparse matrices can occur in our Kronecker approach, clearly extreme cases might be best managed by explicitly storing a list of triplets $(i, j, \mathbf{A}_{i,j})$, one for each nonzero in $\mathbf{A}$.

5.1. The shuffle algorithm.

The first algorithm for the analysis of structured Markov chains was presented in [24]. Fig. 5.1 shows algorithms *Shffl*, to compute $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{x}} \cdot \otimes_{k=1}^K \mathbf{A}^k$, and *Shffl*$^+$, 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).

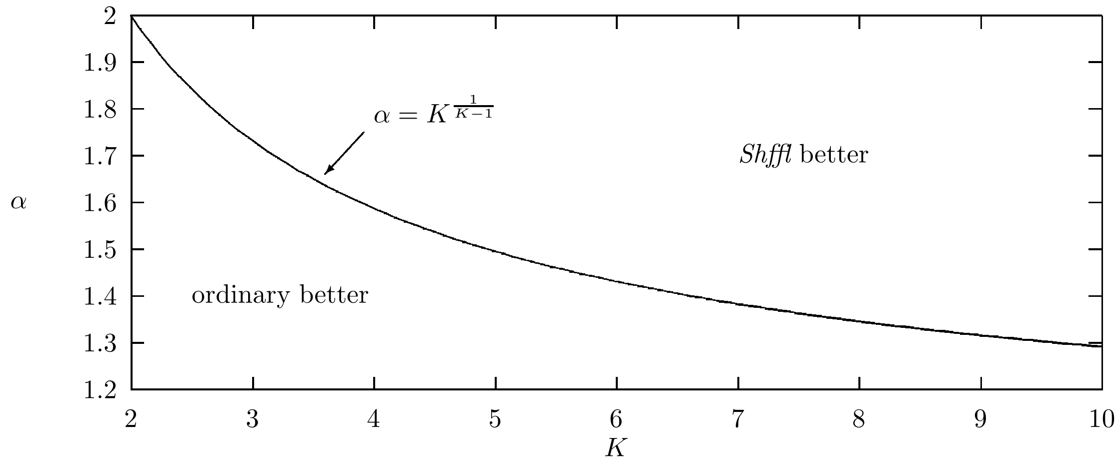


FIG. 5.2. *Comparing Shffl with ordinary multiplication in the (K,α) plane.*

Shffl considers the matrices $\mathbf{A}^k$ sequentially, exploiting the equality [10]:

$$\bigotimes_{k=1}^{K} \mathbf{A}^k = \prod_{k=1}^{K} \mathbf{S}^T_{(n_1^k, n_{k+1}^K)} \cdot (\mathbf{I}_{\bar{n}_k} \otimes \mathbf{A}^k) \cdot \mathbf{S}_{(n_1^k, n_{k+1}^K)}, \tag{5.1}$$

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:

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

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.

The computation of the shuffle permutation is encoded in steps 10–13 and 15–18. The complexity of *Shffl* (derived in [4] as a generalization of the full storage case described in [26, Theorem 9.1]) 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, *Shffl* is faster than a multiplication using explicit storage iff

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

Fig. 5.2 illustrates the regions where *Shffl* or ordinary multiplication perform better, according to the values of K (which is small in practical modeling applications) and α.

Shffl$^+$ is the specialization of *Shffl* 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 (Fig. 5.1). Its complexity is

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

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

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

```
$RwEl$(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.    $j'_1 \leftarrow j_1$;
    3.    $a_1 \leftarrow \mathbf{A}^1_{i_1,j_1}$;
    4.    for each $j_2$ s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
    5.      $j'_2 \leftarrow j'_1 \cdot n_2 + j_2$;
    6.      $a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2}$;
        ...
    7.      for each $j_K$ s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
    8.        $j'_K \leftarrow j'_{K-1} \cdot n_K + j_K$;
    9.        $a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K}$;
    10.       $\hat{\mathbf{y}}_{j'_K} \leftarrow \hat{\mathbf{y}}_{j'_K} + x \cdot a_K$;
```

```
$RwEl^+$(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.    $j' \leftarrow (i^-_k \cdot n_k + j_k) \cdot n^K_{k+1} + i^+_k$;
    3.    $\hat{\mathbf{y}}_{j'} \leftarrow \hat{\mathbf{y}}_{j'} + x \cdot \mathbf{A}^k_{i_k,j_k}$;
```

```
$Rw$(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.    $RwEl(i, \hat{\mathbf{x}}_i, n_{[1,K]}, \mathbf{A}^{[1,K]}, \hat{\mathbf{y}})$;
```

```
$Rw^+$(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.    $RwEl^+(n_k, n^K_{k+1}, i^-_k, i_k, i^+_k, \hat{\mathbf{x}}_i, \mathbf{A}^k, \hat{\mathbf{y}})$;
```

FIG. 5.3. *Vector-matrix multiplication by rows.*

5.2. A straightforward algorithm using sparse storage. A direct application of Eq. (2.1) results in the algorithm *Rw* in Fig. 5.3, 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 *RwEl* computes the contribution of a single entry $\hat{\mathbf{x}}_i$ to all the entries of $\hat{\mathbf{y}}$, as $\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \hat{\mathbf{x}}_i \cdot \mathbf{A}_{i,\hat{\mathcal{T}}}$. The K nested for statements in procedure *RwEl* are required to compute element $\mathbf{A}_{i,j}$ according to its definition in Eq. (2.1). Some of the computation needed to obtain $\mathbf{A}_{i,j}$ is reused for other elements of matrix $\mathbf{A}$. On a given call to *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. *Rw* calls *RwEl* n times, hence its complexity is

$$(5.2)\qquad O\left(n \cdot \sum_{k=1}^{K} \alpha^k\right) = \begin{cases} O(n \cdot K) = O(K \cdot \eta[\mathbf{A}]) & \text{ultrasparse} \\ O(n \cdot \alpha^K) = O(\eta[\mathbf{A}]) & \text{sparse} \end{cases}$$

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 *Cl* and procedure *ClEl*, for multiplication by columns, are omitted. Each call to *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 is equal to $j_{[1,K]}$ in our mixed-base notation. *Cl* has the same complexity as *Rw* but requires sparse column-wise storage for the matrices $\mathbf{A}^k$. However, *Rw* and *Cl* differ in an important aspect: the multiplication performed by *ClEl* requires only one scalar accumulator, while *RwEl* uses the vector $\hat{\mathbf{y}}$ itself as an accumulator. If we follow the good numerical practice of using higher precision for accumulators, *Rw* has larger memory requirements than *Cl*.

The simplified multiplication algorithm Rw^+, used to compute Kronecker sums, is also shown in Fig. 5.3. 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 Rw^+ is then

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

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

TABLE 5.1
Number of nonzeros using the second decomposition.

5.3. Considering only a subsets of states. The decomposition considered so far for our running example satisfies $\hat{\mathcal{T}} = \mathcal{T}$. This is not the case in a second decomposition we now introduce, obtained by further refining the submodels H and L. For the high-priority parts, we define submodels 1, describing the processing of parts of type A and their joint processing with parts of type B, and 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 assembling parts A with B, or C with 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 5.1 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 *Shffl*, *Rw*, and *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 $\mathcal{T}$.

This observation motivates us to 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 $\mathbf{y}$ 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 $\mathbf{x}$: $\hat{\mathbf{x}}_i = \mathbf{x}_I$. To restrict ourselves to reachable states only, we need to:

- Generate $\mathcal{T}$. Efficient algorithms for the generation of $\mathcal{T}$ can be found in [8, 21]
- 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 $\hat{\mathbf{R}}_{\mathcal{T},\hat{\mathcal{T}}\setminus\mathcal{T}} = \mathbf{0}$ [9, 18]. The reverse is 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 $\mathcal{T}$.

Algorithm $RwSb_1$ in Fig. 5.4 modifies Rw, by accessing only elements corresponding to reachable states. We omit the algorithm $RwSb_1^+$ to compute Kronecker sums, since an analogous discussion as for Rw^+

```
$RwElSb_1$(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.   $a_1 \leftarrow \mathbf{A}^1_{i_1,j_1}$;
  3.   for each $j_2$ s.t. $\mathbf{A}^2_{i_2,j_2} > 0$
  4.     $a_2 \leftarrow a_1 \cdot \mathbf{A}^2_{i_2,j_2}$;
     ...
  5.     for each $j_K$ s.t. $\mathbf{A}^K_{i_K,j_K} > 0$
  6.       $a_K \leftarrow a_{K-1} \cdot \mathbf{A}^K_{i_K,j_K}$;
  7.       $J \leftarrow \Psi(j_{[1,K]})$;
  8.       $\mathbf{y}_J \leftarrow \mathbf{y}_J + x \cdot a_K$;
```

```
$RwSb_1$(in: $\mathbf{x}, \mathbf{A}^{[1,K]}, \mathcal{T}$; inout: $\mathbf{y}$)
  1. for each $i_{[1,K]} \in \mathcal{T}$
  2.   $I \leftarrow \Psi(i_{[1,K]})$;
  3.   $RwElSb_1(i_{[1,K]}, \mathbf{x}_I, \mathbf{A}^{[1,K]}, \mathbf{y})$;
```

FIG. 5.4. *Vector-matrix multiplication by rows for a subset $\mathcal{T}$ of the states.*

applies. Line 1 in $RwSb_1$ 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 $RwElSb_1$, 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, a multiplicative overhead factor $O(\log|\mathcal{T}|)$ is encountered in the innermost for-loop. The overall complexity of $RwSb_1$ is then derived analogously to that of Rw. On a given call to $RwElSb_1$, 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. $RwSB_1$ calls $RwElSb_1$ once for each $i_{[1,K]} \in \mathcal{T}$, hence its complexity is

$$O\left(|\mathcal{T}| \cdot \left(\sum_{k=1}^{K} \alpha^k + \alpha^K \cdot \log|\mathcal{T}|\right)\right) = \begin{cases} O\left(|\mathcal{T}| \cdot (K + \log|\mathcal{T}|)\right) & \text{ultrasparse} \\ O\left(|\mathcal{T}| \cdot \alpha^K \cdot \log|\mathcal{T}|\right) & \text{sparse} \end{cases} \tag{5.3}$$

Since $K < \log|\mathcal{T}|$ in practical modeling situations, we can conclude that $RwSb_1$ has a $\log|\mathcal{T}|$ overhead with respect to ordinary multiplication, regardless of the matrix sparsity.

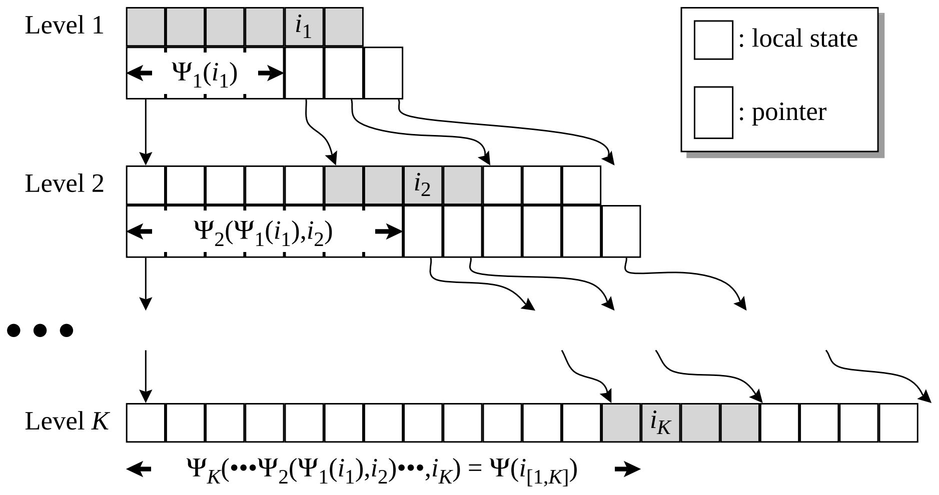
In a multiplication by columns, the situation is potentially worse. $ClSb_1$, the version analogous to $RwSb_1$, must avoid the "spurious" entries in $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$. In $ClElSb_1$, the index $I \leftarrow \Psi(i_{[1,K]})$ computed by the binary search returns null if the "from" state $i_{[1,K]}$ is not reachable. Hence, $ClElSb_1$ 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 longer than for $RwElSb_1$, since searching a state not in $\mathcal{T}$ represents a worst-case scenario, but, more importantly, the complexity of $ClSb_1$ 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 in $RwSb_1$. The sparser $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$ is, the closer the performance of $ClSb_1$ is to that of $RwSb_1$, and the two complexities coincide when $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}} = \mathbf{0}$ (this can happen even when $\hat{\mathcal{T}} \supset \mathcal{T}$).

5.4. Reducing the $\log|\mathcal{T}|$ overhead. The multiplicative overhead $\log|\mathcal{T}|$ in $RwSb_1$ and $ClSb_1$ results from a worst-case assumption that we must search in a set of size $|\mathcal{T}|$ to compute each value of Ψ.

[19] discusses approaches to reduce this overhead, but the most effective solution appears to be obtained by storing $\mathcal{T}$ using the multilevel data structure shown in Fig. 5.5 [8].

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

- $\hat{\mathcal{T}}^k_1 = \mathcal{T}^1 \times \cdots \times \mathcal{T}^k = \{0,\ldots,n_1-1\} \times \cdots \times \{0,\ldots,n_k-1\}$, the projection of the potential state space over the first k components.
- $\mathcal{T}^k_1 = \{i_{[1,k]} : \exists i_{[k+1,K]}, i_{[1,K]} \in \mathcal{T}\}$, the projection of the actual state space over the first k components.



Level 1
i_1
$\Psi_1(i_1)$
: local state
: pointer
Level 2
i_2
$\Psi_2(\Psi_1(i_1),i_2)$
Level K
i_K
$\Psi_K(\cdots\Psi_2(\Psi_1(i_1),i_2)\cdots,i_K) = \Psi(i_{[1,K]})$

FIG. 5.5. *Storage scheme for computation of* Ψ *in* $O(\log|\mathcal{T}|)$.

- $\mathcal{T}^k(i_{[1,k-1]}) = \{i_k : i_{[1,k]} \in \mathcal{T}_1^k\}$, 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.

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. 5.5, 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, $\Psi(i_{[1,K]}) = \mathsf{null}$.

Since 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, $\mathcal{T}$ can be stored in $O(|\mathcal{T}| \cdot \log_2 n_K)$ bits [8].

The real advantage of this data structure, however, 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 *RwSb*$_2$ and *ClSb*$_2$. Fig. 5.6 shows *ClSb*$_2$, since it is used in the solution algorithm *A-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 *ClSb*$_2$, but also for *RwSb*$_2$[1]. In addition, *ClSb*$_2$ still is affected by nonzeros in $\mathbf{A}_{\hat{\mathcal{T}},\mathcal{T}}$, as discussed in Sect. 5.3, requiring the test $I_K \neq \mathsf{null}$ in the innermost loop. The analogous test is instead not needed in *RwSb*$_2$.

[1]We thank A. S. Miner for pointing out this possibility and providing an example where it occurs.

The complexity of $ClElSb_2$ 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 $ClElSb_2$, 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. $ClSb_2$ calls $ClElSb_2$ for each $j_{[1,K]} \in \mathcal{T}$, hence its complexity is

$$(5.4) \qquad O\left(|\mathcal{T}| \cdot \sum_{k=1}^{K} \alpha^k \cdot \log n_k\right) = \begin{cases} O\left(|\mathcal{T}| \cdot \sum_{k=1}^{K} \log n_k\right) = O\left(|\mathcal{T}| \cdot \log n\right) & \text{ultrasparse} \\ O\left(|\mathcal{T}| \cdot \alpha^K \cdot \log n_K\right) & \text{sparse} \end{cases}$$

(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 $RwSb_2$ is analogous, except that the amount of computation performed in $ClSb_2$ is still worse than for $RwSb_2$ because searches for unreachable states are still possible. The problem is now reduced with respect to $ClSb_1$, though, because entries in $\mathbf{A}_{\hat{\mathcal{T}}\setminus\mathcal{T},\mathcal{T}}$ are now discovered before reaching the innermost loop, if $i_{[1,k]} \notin \mathcal{T}_1^k$ for some $k < K$.

Comparing Eq. (5.4) with Eq. (5.3), we conclude that $RwSb_2$ and $ClSb_2$ 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 $RwSb_1$ and $ClSb_1$, $RwSb_2$ and $ClSb_2$ avoid reaching the innermost for-loop whenever a partial state is not reachable, so they might actually perform better than $RwSb_1$ and $ClSb_1$ even in the ultrasparse case.

$RwSb_2^+$ and $ClSb_2^+$ are the simplified algorithms for matrices arising from a Kronecker sum. Fig. 5.6 shows $ClSb_2^+$. On a given call, $ClElSb_2^+$ reaches statements $I_k \leftarrow \Psi_k(I_{k-1}, i_k)$ and $I_h \leftarrow \Psi_h(I_{h-1}, j_h)$ for $k < h \leq K$, 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. $ClSb_2^+$ calls $ClElSb_2^+$ 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 $ClSb_2^+$ 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 $|\mathcal{T}| \cdot \alpha \cdot K$.

5.5. Interleaving rows and columns. $RwSb_2$ and $ClSb_2$ fail to amortize the logarithmic searches in the case of ultrasparse matrices because the K nested for-loops in $RwElSb_2$ and $ClElSb_2$ 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 need to consider the elements of $\mathbf{A}$ in a different order, interleaving the K nested for-loops in $RwElSb_2$, or $ClElSb_2$, with those implicitly defined by the single for-loop in $RwSb_2$, or $ClSb_2$.

```
$ClElSb_2$(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.               $y \leftarrow y + \mathbf{x}_{I_K} \cdot a_K$;
```

```
$ClSb_2$(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.   $ClElSb_2(j_{[1,K]}, \mathbf{x}, \mathbf{A}^{[1,K]}, \mathbf{y}_J)$;
```

```
$RwSb_3$(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.                     $\mathbf{y}_{J_K} \leftarrow \mathbf{y}_{J_K} + \mathbf{x}_{I_K} \cdot a_K$;
```

```
$ClElSb_2^+$(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.         $y \leftarrow y + \mathbf{x}_{I_K} \cdot \mathbf{A}^k_{i_k,j_k}$;
```

```
$ClSb_2^+$(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.   $ClElSb_2^+(j_{[1,K]}, \mathbf{x}, k, \mathbf{A}^k, \mathbf{y}_J)$;
```

```
$RwSb_3^+$(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.           $\mathbf{y}_{J_K} \leftarrow \mathbf{y}_{J_K} + \mathbf{x}_{I_K} \cdot \mathbf{A}^k_{i_k,j_k}$;
```

FIG. 5.6. *A better vector-matrix multiplication for a subset of the states.*

We employ this idea to derive algorithm $RwSb_3$, shown in Fig. 5.6 and used in the solution algorithm *A-JCB* of Sect. 6. 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 statement. A search is performed only to obtain $J_k \leftarrow \Psi_k(J_{k-1}, j_k)$ and, even if this is a multiplication by rows, the tests $J_k \neq$ null are necessary, as already discussed for $RwSB_2$ and $ClSb_2$.

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

Procedure	(a) mult	(a) CPU	(a) wall	(b) mult	(b) CPU	(b) wall
Shffl	22,638,318	54.9	178.2	—	—	—
Rw	18,832,668	38.6	43.4	—	—	—
Cl	18,832,668	54.1	54.2	243,941,600	1,488.2	3,325.2
$RwSb_1,\ RwSb_1^+$	18,832,668	156.5	157.5	23,199,652	219.8	220.2
$ClSb_1,\ ClSb_1$	18,832,668	191.4	191.9	23,199,652	259.8	260.5
$RwSb_2,\ RwSb_2^+$	17,987,499	56.3	57.2	20,980,675	60.3	61.8
$ClSb_2,\ ClSb_2$	17,987,499	91.1	91.6	20,980,675	106.7	106.9
$RwSb_3,\ RwSb_3^+$	15,714,589	43.0	45.2	15,960,012	24.4	24.6

TABLE 5.2

Computational effort for vector-matrix multiplication.

the complexity of $RwSb_3$ is:

$$O\left(\sum_{k=1}^{K} |\mathcal{T}_1^k| \cdot \alpha^k \cdot \log n_k\right) = O(|\mathcal{T}| \cdot \alpha^K \cdot \log n_K) \tag{5.5}$$

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

$RwSb_3^+$ in Fig. 5.6 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 $RwSb_3^+$ 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 $RwSb_3^+$ 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.

The complexity of $ClSb_3$ and $ClSb_3^+$ is the same as that of $RwSb_3$ and $RwSb_3^+$, although spurious entries are still a disadvantage. Unfortunately, though, $ClSb_3$, unlike $ClSb_1$ and $ClSb_2$, does not compute the entries of $\mathbf{y}$ in order. This property prevents us from interleaving rows and column indices to reduce the logarithmic overhead if we want to use a Gauss-Seidel type iteration in our solution algorithm.

5.6. Comparing the multiplication algorithms. Fig. 5.7 compares the theoretical complexities of *Shffl* and *Rw* or *Col*, and of ordinary multiplication, assuming $\hat{\mathcal{T}} = \mathcal{T}$ and $K = 4$. While it is clear that *Shffl* is much superior for large values of α, the matrices $\mathbf{W}^k(e)$ in typical modeling applications are ultrasparse or even hypersparse. In the region around $\alpha = 1$, *Shffl* and *Rw* have the same performance, K times worse than ordinary multiplication. Indeed, *Rw* outperforms *Shffl* when $\alpha < 1$, since it may recognize that an entire row of $\mathbf{A}$ is zero before reaching the innermost for-loop.

On the other hand, $Shffl^+$ and Rw^+ 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

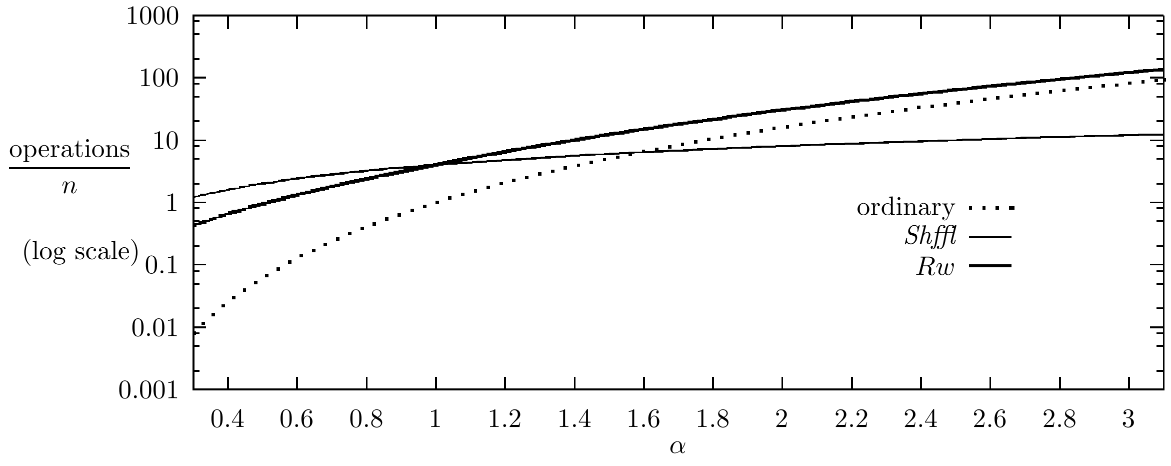


FIG. 5.7. *Comparing the complexity of Shffl, Rw, and ordinary multiplication ($K = 4$).*

(since no partial result can be reused the way *Shffl* 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 $\bigoplus_{k=1}^{K} \mathbf{A}^k$. In our application, the diagonals of the matrices $\mathbf{R}^k$ are zero, so this does not happen.

These observations are confirmed by our running example. Table 5.2 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. (4.1). 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 SPARCstation 4 under SunOS 4.1.4, with a 85 MHz CPU, 64 MB main memory, and virtual memory limited to 398 MB.

In the first decomposition, the matrices $\mathbf{W}^k(e)$ are ultrasparse or hypersparse and, as predicted by our theoretical observations for $\alpha < 1$, *Rw* outperforms *Shffl*, but all Kronecker-based algorithms are less computationally efficient than a conventional multiplication where $\mathbf{R}$ is stored in sparse format, which would only require $\eta[\mathbf{R}] = 13{,}439{,}073$ multiplications.

This suggests that, in practice, the real advantage of Kronecker-based methods lies exclusively in their large memory savings.

In the second decomposition, $\eta[\mathbf{A}_{\hat{\mathcal{T}},\mathcal{T}}] = \eta[\mathbf{A}_{\mathcal{T},\mathcal{T}}]$, hence one should expect no significant difference between row and column algorithms. However, as discussed in the following section, we use column algorithms only to allow a Gauss-Seidel type of iteration, where entries of the new iterate of the steady-state probability vector must be computed sequentially. Hence, 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 are free to consider one event at a time, and all the states it can affect. The latter way avoids having to switch between events, hence the row algorithms perform better.

However, the column variants use less memory because the operand vector is directly overwritten by the results. This is the reason why, for the second decomposition, *Cl* can still execute while *Shffl* and *Rw* fail due to excessive memory requirements. However, *Cl* heavily relies on virtual memory, as the difference between CPU and elapsed times indicates. *Cl* considers 139,172,250 matrix entries in $\hat{\mathbf{R}}$, although only

$\eta[\mathbf{R}] = 13{,}439{,}073$ are relevant.

Shffl, *Rw*, and *Cl* are obviously inadequate when $|\hat{\mathcal{T}}| \gg |\mathcal{T}|$; only algorithms based on $\mathcal{T}$ run acceptably fast for the second decomposition. The results indicate that their overhead is effectively reduced from $RwSb_1$ to $RwSb_2$ to $RwSb_3$, and from $ClSb_1$ to $ClSb_2$. Clearly, there is no reason ever to use $RwSb_1$ or $ClSb_1$; we introduced them only as a stepping stone to the better algorithms.

In summary, $RwSb_3$ is a fast and robust algorithm, almost as fast as *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 $\mathcal{T}$. For multiplication by columns, instead, *Cl* is considerably faster than $ClSb_2$ when $|\mathcal{T}| = |\hat{\mathcal{T}}|$, but $ClSb_2$ 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 $RwSb_3$ 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: in the first decomposition, n_K is much larger than in the second one (2394 vs. 126); indeed, $43.0/24.4 = 1.76 \approx \log 2394/\log 126 = 1.61$. 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 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 [20] .

6. Model solution algorithms. We now return to the problem of solving a Markov model, that is, Eq. (1.1). In practical modeling problems, $\mathbf{Q}$ is very large and indirect iterative numerical methods such as Power, Jacobi, or Gauss-Seidel are normally employed for the solution. In all cases, starting from a guess $\boldsymbol{\pi}^{(0)}$, which does not have to be the initial probability vector if the CTMC is ergodic, 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, $\max_{0 \le I < |\mathcal{T}|}\{\mathbf{h}_I\}$. Element-wise, the Power method corresponds to:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \boldsymbol{\pi}_J^{(m)} + \left(\sum_{0 \le I < |\mathcal{T}|} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{Q}_{I,J} \right) \cdot h^*. \tag{6.1}$$

  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:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \left(\sum_{0 \le I < |\mathcal{T}|, I \ne J} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{R}_{I,J} \right) \cdot \mathbf{h}_J. \tag{6.2}$$

  The Jacobi method does not have guaranteed convergence, but it is usually faster than the Power method in practice. The Jacobi and Power methods coincide when all the sojourn times have the same value and, in the Power method, h^* is set to this value instead of a value slightly larger.

- 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:

$$\forall J \in \{0, 1, \ldots |\mathcal{T}| - 1\}, \quad \boldsymbol{\pi}_J^{(m+1)} \leftarrow \left(\sum_{0 \le I < J} \boldsymbol{\pi}_I^{(m+1)} \cdot \mathbf{R}_{I,J} + \sum_{J < I < |\mathcal{T}|} \boldsymbol{\pi}_I^{(m)} \cdot \mathbf{R}_{I,J} \right) \cdot \mathbf{h}_J. \tag{6.3}$$

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.

Relaxation can be used to accelerate convergence [27]. We do not investigate this possibility, since it does not affect our discussion. Other iterative solution techniques, such as projection methods, have also been applied successfully for the analysis of Kronecker-based models. However, these techniques have higher memory requirements and, in any case, they too perform vector-matrix multiplications, so our discussion could be extended to them. For a detailed analysis of numerical techniques for the solution of Markov chains, see [26].

6.1. Alternative solution approaches. 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 [13, 14, 25], 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.

Previous approaches, however, employ only the Power or Jacobi methods because they restrict themselves to accessing the matrix $\mathbf{R}$ by rows. As pointed out in Sect. 5, they compute the entries of a new iterate $\boldsymbol{\pi}^{(m+1)}$ incrementally, using the values of the previous iterate $\boldsymbol{\pi}^{(m)}$, so that double-precision vectors should be used.

The use of Gauss-Seidel requires instead computing $\boldsymbol{\pi}^{(m+1)}_{\{1,\ldots I-1\}}$ before $\boldsymbol{\pi}^{(m+1)}_{I}$. This can be accomplished if we have access to $\mathbf{R}$ by columns, that is, if we can efficiently obtain all the nonzero entries $\hat{\mathbf{R}}_{i,j}$, for a given $j \in \mathcal{T}$ and any $i \in \mathcal{T}$. We have shown how to do this in Sect. 5. An additional advantage is that single-precision vectors can be used in this case.

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

- Use the perfect shuffle approach, *SH*, or our multiplication procedures.
- Store vectors the size of the potential, *P*, or actual, *A*, state space.
- Perform a Jacobi (*JCB*) or Gauss-Seidel (*GSD*) iteration.

We indicate the resulting algorithms as *SH-JCB* [4, 15], *P-JCB*, *A-JCB*, *P-GSD*, and *A-GSD*. In the original SAN paper [24] introducing the Kronecker-based solution approach, the Power method is used instead of Jacobi. Thus, we present first the Jacobi method using function *Shffl* to realize the iteration. Fig. 6.1 and 6.2 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 $\mathbf{W}^k(e)$, for some k (in a practical implementation, it is best to choose a k for which $\mathbf{W}^k(e) \neq \mathbf{I}$).

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 [12] have shown how to use a variant of Gauss-Seidel in conjunction with an on-the-fly approach

```
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.  $\hat{\boldsymbol{\pi}}^{aux2} \leftarrow \mathbf{0}$;
    3.  foreach $e \in \mathcal{E}_S$
    4.      $\hat{\boldsymbol{\pi}}^{aux1} \leftarrow \hat{\boldsymbol{\pi}}^{old}$;
    5.      Shffl($n_{[1,K]}, \mathbf{W}^{[1,K]}(e), \hat{\boldsymbol{\pi}}^{aux1}, \hat{\boldsymbol{\pi}}^{aux2}$);
    6.      $\hat{\boldsymbol{\pi}}^{new} \leftarrow \hat{\boldsymbol{\pi}}^{new} + \hat{\boldsymbol{\pi}}^{aux2}$;
    7.  for $k = 1$ to $K$
    8.      Shffl$^{+}$($\hat{\boldsymbol{\pi}}^{old}, n_1^{k-1}, n_k, n_{k+1}^K, \mathbf{R}^k, \hat{\boldsymbol{\pi}}^{new}$);
    9.  for $i = 0$ to $n_1^K - 1$
    10.     if $\hat{\boldsymbol{\pi}}^{new}_i > 0$ then
    11.         $\hat{\boldsymbol{\pi}}^{new}_i \leftarrow \hat{\boldsymbol{\pi}}^{new}_i \cdot \hat{\mathbf{h}}_i$;
```

FIG. 6.1. *Algorithm SH-JCB.*

for a set of modeling formalisms including GSPNs, stochastic activity networks, and stochastic reward nets. A similar idea is also in [22]. However, the time complexity of this approach 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, since entire paths, not single events, must be explored in this case to generate a single entry, so we do not consider it further.

6.2. Algorithm *SH-JCB***.** The algorithm *SH-JCB* in Fig. 6.1 implements the Jacobi method using *Shffl* and *Shffl*$^{+}$ for the vector-matrix multiplications. The time complexity of one *SH-JCB* iteration is independent of the submodel ordering:

$$O\left(\sum_{k=1}^{K} \bar{n}_k \cdot \left(\eta[\mathbf{R}^k] + \sum_{e:\mathbf{W}^k(e)\neq\mathbf{I}} \eta[\mathbf{W}^k(e)]\right)\right). \tag{6.4}$$

Five vectors of length n are needed: 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 two auxiliary vectors used when calling procedure *Shffl*, $\hat{\boldsymbol{\pi}}^{aux1}$ and $\hat{\boldsymbol{\pi}}^{aux2}$. Additionally, two vector $\mathbf{z}$ and $\mathbf{z}'$ are needed in the procedures *Shffl* and *Shffl*$^{+}$, but they are only of size $\max_k(n_k)$, much smaller than n. Of these, $\hat{\boldsymbol{\pi}}^{new}$, $\hat{\boldsymbol{\pi}}^{aux2}$, and $\mathbf{z}'$ should be stored in double-precision, because they are used as accumulators.

6.3. Algorithm *P-JCB***.** *P-JCB* is the simplest iteration, it uses the *Rw* and *Rw*$^{+}$ vector-matrix multiplications presented in Fig. 5.3 and, just as algorithm *SH-JCB*, it uses vectors of length n. Its complexity depends on the order of the components:

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

P-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 *A-JCB***.** *A-JCB* has the same convergence behavior of *SH-JCB* and *P-JCB*, but uses data structures of size $|\mathcal{T}|$ by employing the $RwSb_3$ and $RwSb_3^{+}$ vector-matrix multiplications presented in Fig. 5.6. The complexity of one *A-JCB* iteration is

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

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

```
$A\text{-}JCB$(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.  foreach $e \in \mathcal{E}_S$
    3.    $RwSb_3(\boldsymbol{\pi}^{old}, \mathbf{W}^{[1,K]}(e), \mathcal{T}, \boldsymbol{\pi}^{new})$;
    4.  for $k = 1$ to $K$
    5.    $RwSb_3^{+}(\boldsymbol{\pi}^{old}, k, \mathbf{R}^k, \mathcal{T}, \boldsymbol{\pi}^{new})$;
    6.  for $I = 0$ to $|\mathcal{T}| - 1$
    7.    $\boldsymbol{\pi}_I^{new} \leftarrow \boldsymbol{\pi}_I^{new} \cdot \mathbf{h}_I$;
```

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

```
$A\text{-}GSD$(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.    $J \leftarrow \Psi(j_{[1,K]})$;
    3.    $\boldsymbol{\pi}_J \leftarrow 0$;
    4.    foreach $e \in \mathcal{E}_S$
    5.      $ClElSb_2(j_{[1,K]}, \boldsymbol{\pi}, \mathbf{W}^{[1,K]}(e), \boldsymbol{\pi}_J)$;
    6.    for $k = 1$ to $K$
    7.      $ClElSb_2^{+}(j_{[1,K]}, \boldsymbol{\pi}, k, \mathbf{R}^k, \boldsymbol{\pi}_J)$;
    8.    $\boldsymbol{\pi}_J \leftarrow \boldsymbol{\pi}_J \cdot \mathbf{h}_J$;
```

FIG. 6.2. *Algorithms P-JCB, A-JCB, P-GSD, and A-GSD.*

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 *A-JCB* are the same as for *P-JCB*, except that vectors are now of size $|\mathcal{T}|$, not n.

6.5. Algorithm *P-GSD*. With the Gauss-Seidel method, the old and the new iterate can be stored into a single vector. If $\mathbf{R}$ were described by a single Kronecker product $\bigotimes_{k=1}^{K} \mathbf{A}^k$, *P-GSD* would be achieved by the simple call $Cl(\hat{\boldsymbol{\pi}}, n_{[1,K]}, \mathbf{A}^{[1,K],l}, \hat{\boldsymbol{\pi}})$, followed by the same elementwise multiplication of $\hat{\boldsymbol{\pi}}$ by the expected

Procedure	**iteration vectors**	**holding time vector**	**auxiliary vectors**	**search data structure**
SH-JCB	$n\cdot$ (S+D)	$n\cdot$ S	$n\cdot$ (S+D)	—
P-JCB	$n\cdot$ (S+D)	$n\cdot$ S	—	—
A-JCB	$\lvert\mathcal{T}\rvert\cdot$ (S+D)	$\lvert\mathcal{T}\rvert\cdot$ S	—	$\approx \lvert\mathcal{T}\rvert\cdot$ L
P-GSD	$n\cdot$ S	$n\cdot$ S	—	—
A-GSD	$\lvert\mathcal{T}\rvert\cdot$ S	$\lvert\mathcal{T}\rvert\cdot$ S	—	$\approx \lvert\mathcal{T}\rvert\cdot$ L

Table 6.1
Memory requirements for model solution algorithms.

holding times, as performed by *P-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 the new $\hat{\boldsymbol{\pi}}_i$ before starting that of $\hat{\boldsymbol{\pi}}_{i+1}$. Hence, *P-GSD* must call the functions *ClEl* and *ClEl*$^+$ directly, not through *Cl* or *Cl*$^+$.

The complexity of *P-GSD* is then the same as that of *P-JCB*. This makes it a better choice, since Gauss-Seidel has better convergence than Jacobi, and 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 ***A-GSD*****.** The comments made for *P-GSD* apply to *A-GSD* as well. As observed at the end of Sect. 5, the interleaving or rows and columns of *ClSb*$_3$ and *ClSb*$_3^+$ cannot be used, so *ClSb*$_2$ and *ClSb*$_2^+$ must be used instead, whose amortization of the logarithmic search is less effective. This points out a surprising tradeoff between *A-JCB*, which has slower convergence but a smaller overhead, $\log n_K$, and *A-GSD*, which has better numerical convergence but higher overhead, possibly as high as $\log n$.

The complexity of *A-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 6.1 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, 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 6.2, 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 6.3. We performed iterations using the absolute convergence criterion $||\boldsymbol{\pi}^{old}-\boldsymbol{\pi}^{new}||_\infty < 10^{-8}$, and a relaxation factor of 0.95.

As already anticipated in Table 5.2, algorithms *SH-JCB* and *P-JCB* fail due to insufficient memory with the second decomposition, while *P-GSD* could be run, but with an unacceptable amount of overhead (we estimate it would require about six days of CPU time).

We observe that the two decompositions result in different state orderings, which in turn affect the convergence of *A-GSD*. Hence, 200 iterations are required for the first decomposition, but only 150 for the second one (convergence is tested every 10 iterations).

Procedure	**(a) vectors**	**(a) extra**	**(b) vectors**	**(b) extra**
SH-JCB	24,974,208	18,730,656	276,595,200	207,446,400
P-JCB	24,974,208	—	276,595,200	—
A-JCB	24,974,208	3,127,000	24,974,208	3,804,700
P-GSD	12,487,104	—	138,297,600	—
A-GSD	12,487,104	3,127,000	12,487,104	3,804,700

Table 6.2
Memory requirements (in bytes) for our example.

Procedure	**(a) CPU**	**(a) wall**	**(a) iter**	**(b) CPU**	**(b) wall**	**(b) iter**
SH-JCB	20,327	65,940	370	—	—	—
P-JCB	14,292	16,067	370	—	—	—
A-JCB	15,928	16,725	370	10,022	10,054	370
P-GSD	10,822	10,841	200	—	—	—
A-GSD	18,236	18,337	200	20,464	21,218	150

Table 6.3
Execution times (seconds) and number of iterations for our example.

7. Conclusion.

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

We have shown how the Kronecker-based solution of structured Markov models can be carried on 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 allowing the use of both Jacobi- and Gauss-Seidel-style methods.

Our results are not limited to steady-state 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 [20] for the Petri net formalism; implementation of a more general software package fully supporting the state-dependent behavior we described is under way [7]. 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 (1984), pp. 93–122.

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

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

[4] ——, *A class of hierarchical queueing networks and their analysis*, Queueing Systems., 15 (1994), pp. 59–80.

[5] G. Chiola, *On the structural and behavioral characterization of P/T nets*, in Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM'93), Toulouse, France, Oct. 1993, IEEE Comp. Soc. Press.

[6] 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 Linear Algebra, Markov Chains, and Queueing Models, C. Meyer and R. J. Plemmons, eds., vol. 48 of IMA Volumes in Mathematics and its Applications, Springer-Verlag, 1993, pp. 145–191.

[7] 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), Urbana-Champaign, IL, USA, Sept. 1996, IEEE Comp. Soc. Press, p. 60.

[8] ——, *Storage alternatives for large structured state spaces*, in Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, eds., LNCS 1245, Saint Malo, France, June 1997, Springer-Verlag, pp. 44–57.

[9] 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.

[10] M. Davio, *Kronecker products and shuffle algebra*, IEEE Trans. Comp., C-30 (1981), pp. 116–125.

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

[12] ——, *"On-the-fly" solution techniques for stochastic Petri nets and extensions*, in Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97), Saint Malo, France, June 1997, IEEE Comp. Soc. Press, pp. 132–141.

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

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

[15] P. Fernandes, B. Plateau, and W. J. Stewart, *Efficient descriptor-vector multiplication in stochastic automata networks*, Rapport Apache (LGI, LMC) 12, 1994.

[16] U. Herzog and M. Rettelbach, eds., *Proc. of the 2nd Workshop on Process Algebra and Performance Modelling (PAPM)*, Arbeitsberichte des IMMD 27 (4), 1994, IMMD, Universität Erlangen.

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

[18] P. Kemper, *Numerical analysis of superposed GSPNs*, IEEE Trans. Softw. Eng., 22 (1996), pp. 615–628.

[19] ——, *Superposition of generalized stochastic Petri nets and its impact on performance analysis*, PhD

thesis, Universität Dortmund, 1996.

[20] ———, *SupGSPN Version 1.0 - an analysis engine for superposed GSPNs*, tech. rep., Universität Dortmund, 1997.

[21] ———, *Reachability analysis based on structured representations*, in Application and Theory of Petri Nets 1996, Lecture Notes in Computer Science 1091 (Proc. 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka, Japan), J. Billington and W. Reisig, eds., Springer-Verlag, June 1999, pp. 269–288.

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

[23] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press, 1984.

[24] 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, Austin, TX, USA, May 1985, pp. 147–153.

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

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

[27] W. J. Stewart and A. Goyal, *Matrix methods in large dependability models*, Tech. Rep. RC-11485, IBM T.J. Watson Res. Center, Yorktown Heights, NY, Nov. 1985.