SIAM J. SCI. COMPUT.
Vol. 37, No. 5, pp. S526–S543

ON VECTOR-KRONECKER PRODUCT MULTIPLICATION WITH RECTANGULAR FACTORS*

TUĞRUL DAYAR† AND M. CAN ORHAN†

Abstract. The infinitesimal generator matrix underlying a multidimensional Markov chain can be represented compactly by using sums of Kronecker products of small rectangular matrices. For such compact representations, analysis methods based on vector-Kronecker product multiplication need to be employed. When the factors in the Kronecker product terms are relatively dense, vector-Kronecker product multiplication can be performed efficiently by the shuffle algorithm. When the factors are relatively sparse, it may be more efficient to obtain nonzero elements of the generator matrix in Kronecker form on the fly and multiply them with corresponding elements of the vector. This work proposes a modification to the shuffle algorithm that multiplies relevant elements of the vector with submatrices of factors in which zero rows and columns are omitted. This approach avoids unnecessary floating-point operations that evaluate to zero during the course of the multiplication and possibly reduces the amount of memory used. Numerical experiments on a large number of models indicate that in many cases the modified shuffle algorithm performs a smaller number of floating-point operations than the shuffle algorithm and the algorithm that generates nonzeros on the fly, sometimes with a minimum number of floating-point operations and as little of memory possible.

Key words. Markov chain, Kronecker representation, vector-Kronecker product multiplication, shuffle algorithm

AMS subject classifications. 60J27, 65F50, 15A72, 65F10, 65B99

DOI. 10.1137/140980326

1. Introduction.

Markov chains (MCs) are state and transition based probabilistic models widely used to analyze the behavior of systems arising in different application areas. When they aid the modeling and analysis of systems composed of interacting subsystems [31], a multidimensional state representation in which each subsystem normally corresponds to a different dimension may be utilized. For a given transition triggered by an event in this representation, the state of a subsystem may either evolve independently or evolve in synchronization with other subsystems. With this understanding, the reachable state space of a multidimensional MC is the set of states which the system is able to reach by following the possible transitions that take place due to different events. In many cases, the semantics of the system dictates that the reachable state space of the MC model be a proper subset of its product state space, that is, the Cartesian product of the subsystem state spaces.

The multidimensional state representation together with the Cartesian product operator motivate the use of Kronecker products [33] of the smaller transition matrices associated with subsystems in defining the larger infinitesimal generator matrix underlying the MC. This approach enables the relatively easy specification of multidimensional MCs and the compact representation of their generator matrices [16]. Since the generator matrix needs to be kept in compact form during analysis, an efficient vector-Kronecker product multiplication algorithm based on shuffle algebra

*Received by the editors August 1, 2014; accepted for publication (in revised form) June 15, 2015; published electronically October 29, 2015.
http://www.siam.org/journals/sisc/37-5/98032.html
†Department of Computer Engineering, Bilkent University, TR-06800 Bilkent, Ankara, Turkey (tugrul@cs.bilkent.edu.tr, morhan@cs.bilkent.edu.tr). The research of the second author was supported by the Scientific and Technological Research Council of Turkey.

[15] and known as the shuffle algorithm [28] has been devised and used. In this algorithm, the multiplication of a vector with the Kronecker product is achieved by multiplying the vector with as many matrices as the number of factors, say, H, in the Kronecker product, where each of the H matrices is expressed as the Kronecker product of $(H-1)$ identity matrices and a different factor whose order among the identity matrices is that of the same factor in the original Kronecker product, hence the term "shuffle". Although this approach does not alleviate the problem of having to store large state probability vectors during analysis, it is still more memory efficient than conventional techniques that use sparse vector-matrix multiplication. Regarding time complexity, the shuffle algorithm is slower unless the factors in the Kronecker product become relatively dense.

In Kronecker based Markovian modeling formalisms such as stochastic automata networks [28, 29], the set of rows and the set of columns are each equal to the product state space, implying square factors in the Kronecker form. Consequently, the generator matrix can be expressed as a sum of Kronecker products of subsystem transition matrices, and vector-Kronecker product multiplication algorithms [6, 13, 20, 21] are implemented using an auxiliary flag vector as long as the product state space to indicate the reachability of states and avoid unnecessary floating-point operations (flops) with unreachable states. Clearly these approaches pose considerable inefficiency due to indexing and addressing during analysis when the reachable state space is significantly smaller than the product state space.

Compact storage of the generator matrix underlying an MC with unreachable states and efficient implementation of analysis methods using Kronecker operations require the reachable state space to be represented as a union of Cartesian products of subsets of subsystem state spaces [16], as has been done, for instance, in hierarchical Markovian models [7]. Consequently, the generator matrix can be expressed as a block matrix in which the sets of rows and columns corresponding to each diagonal block are the same, each set equal to a particular partition of the reachable state space, and each block is a sum of Kronecker products of subsystem transition submatrices. The off-diagonal blocks of this matrix need not be square, and hence the shuffle algorithm is implemented for rectangular factors in this case [16]. The hierarchical representation of the generator matrix with the shuffle algorithm for rectangular factors has been successfully used in block iterative methods [9], preconditioned projection methods [11], and multilevel methods [10, 12] for analyzing many different problems. Any improvement in the shuffle algorithm for rectangular factors will translate to an improvement in analysis methods for multidimensional MCs in Kronecker form. It is evident that all square factors is just a special case of rectangular factors; hence, the algorithms considered here are for the general case.

Vector-Kronecker product multiplication algorithms that are of a different nature have been proposed in [13]. In the algorithms therein, nonzero elements of the generator matrix are obtained on the fly and multiplied with the corresponding elements of the vector. These algorithms are devised for square factors, but they can be extended to handle rectangular factors without much difficulty. Pot-RwCl is the most efficient among the algorithms in [13] since it fully exploits common nonzero factors forming the nonzero elements of the sparse matrix corresponding to each Kronecker product term. Due to this, Pot-RwCl becomes more efficient than the shuffle algorithm when the factors in the Kronecker product terms are relatively sparse.

In this work, we aim at improving the efficiency of the shuffle algorithm for rectangular factors. To this end, we consider Kronecker based MCs with given Cartesian product partitionings of their reachable state spaces [16, 17]. We implement the shuffle

algorithm in such a way that zero rows and columns in factors of Kronecker product terms are omitted during multiplication. This enables us to avoid flops that evaluate to zero during the course of the multiplication and to possibly reduce the amount of memory used. The shuffle algorithm, the Pot-RwCl algorithm, and the modified shuffle algorithm are compared analytically for their flop and memory requirements. Although the modification on the shuffle algorithm may seem simple, it is not intuitive (since zero rows and columns are not stored in sparse matrices anyway), but as we shall see through numerical experiments, it turns out to be quite effective in many cases. At the end, we are able to identify those implementations of vector-Kronecker product multiplication that should be preferred over others.

Note that we treat the Kronecker product factors as purely algebraic quantities and do not use their stochastic properties. Therefore, the proposed modification on the shuffle algorithm is not limited to the context of MCs. Moreover, the idea of avoiding some flops by omitting zero rows and columns in factors of Kronecker product terms is not limited to the improvement on the shuffle algorithm. A potential use of the proposed approach is in the semi-tensor product operation that was introduced recently to generalize matrix multiplication [14]. The semi-tensor product of two matrices is defined as the multiplication of Kronecker products of identity matrices and these factors. It should be possible to devise an algorithm based on shuffle algebra to multiply a vector with a semi-tensor product of matrices without computing the semi-tensor product of the matrices. When this is the case, unnecessary flops can be avoided by omitting zero rows and columns in the factors during the course of the multiplication.

Throughout the paper, calligraphic uppercase letters are used for sets. $|\cdot|$, $\times$, and $\otimes$ respectively stand for the number of elements in a set, the Cartesian product operator, and the Kronecker product operator. All vectors are row vectors and are represented with boldface lowercase letters with the exception that $\mathbf{e}$ and $\mathbf{e}_r$ respectively represent a column vector of ones and the rth column of the identity matrix with their lengths being determined from the contexts in which they are used. We start indices from 0, by which it is possible to represent an empty (sub)system. An exception is state vectors, whose components are state variables indicated by subscripts starting from 1. I_r and $\mathrm{diag}(\mathbf{a})$ denote the identity matrix of order r and the diagonal matrix with the entries of vector $\mathbf{a}$ along its diagonal, respectively. $\mathbf{a}(x)$ is the value of vector $\mathbf{a}$ at state x and $\mathbf{a}(\mathcal{X})$ is the subvector of $\mathbf{a}$ associated with the states in $\mathcal{X}$. $A(x,y)$ is the value of matrix A corresponding to element (x,y) and $A(\mathcal{X},\mathcal{Y})$ is the submatrix of A associated with row states in $\mathcal{X}$ and column states in $\mathcal{Y}$. $nnz(A)$ denotes the number of nonzeros in A. Finally, $\mathbb{R}$ and $\mathbb{Z}_{\geq 0}$ denote the sets of reals and nonnegative integers, respectively.

The next section provides background information regarding the Kronecker representation of multidimensional MCs without unreachable states. It provides the shuffle and Pot-RwCl algorithms for rectangular factors and elaborates on how they work using a small example. The third section introduces the proposed modification on the shuffle algorithm. The merit of this approach is shown on the same small example with the help of a lemma. The fourth section discusses implementation issues associated with shuffle, Pot-RwCl, and modified shuffle algorithms. The fifth section presents numerical results on a number of benchmark models and comments on them. The final section concludes the paper.

2. Background. The Kronecker (or tensor) product [15, 33] of two (rectangular) matrices A and B with $A = [A(x_A, y_A)]$ is

$$A \otimes B = [A(x_A, y_A)B] \, .$$

Or more formally, given $A \in \mathbb{R}^{r_A \times c_A}$ and $B \in \mathbb{R}^{r_B \times c_B}$, $A \otimes B$ yields the (rectangular) matrix $C \in \mathbb{R}^{r_A r_B \times c_A c_B}$ whose entries satisfy

$$C(x_C, y_C) = A(x_A, y_A)B(x_B, y_B)$$

$$\text{with} \quad x_C = x_A r_B + x_B \quad \text{and} \quad y_C = y_A c_B + y_B$$

for

$$(x_A, y_A) \in \{0, \ldots, r_A - 1\} \times \{0, \ldots, c_A - 1\} ,$$
$$(x_B, y_B) \in \{0, \ldots, r_B - 1\} \times \{0, \ldots, c_B - 1\} .$$

The Kronecker product is associative and defined for more than two matrices.

Now, let us consider a Markovian system with H interacting subsystems, where $\mathcal{S}_h$ denotes the state space of subsystem $h = 1, \ldots, H$. Without loss of generality, let us assume that subsystem state spaces are defined on consecutive nonnegative integers starting from 0. Otherwise, we can always enumerate subsystem state spaces so that they satisfy this assumption. We denote the reachable state space of the system by $\mathcal{S} \subseteq \times_{h=1}^{H} \mathcal{S}_h$, where $\times_{h=1}^{H} \mathcal{S}_h$ is the product state space. Many times when implementing an algorithm, one needs to have a mapping from the multidimensional reachable state space to the one-dimensional reachable state space, and vice versa, since vectors are one-dimensional and suitable elements of them need to be accessed. In our case, we do have such a mapping. Therefore, multidimensional and one-dimensional representations of states will be used interchangeably. Having said this, we define the Cartesian product partitioning of $\mathcal{S}$ as in [17].

DEFINITION 2.1. *Let $\mathcal{S}^{(i)} = \times_{h=1}^{H} \mathcal{S}_h^{(i)}$, where $\mathcal{S}_h^{(i)}$ is set of consecutive integers for $i = 1, \ldots, J$. Then $\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(J)}$ is a Cartesian product partitioning of $\mathcal{S}$ if $\mathcal{S} = \cup_{i=1}^{J} \mathcal{S}^{(i)}$ and $\mathcal{S}^{(i)} \cap \mathcal{S}^{(j)} = \emptyset$ for $i \neq j$ and $i, j = 1, \ldots, J$.*

The infinitesimal generator matrix Q underlying the MC can be perceived as a block matrix induced by the Cartesian product partitioning of $\mathcal{S}$. Then Q is a $(J \times J)$ block matrix as in

$$Q = \begin{bmatrix} Q^{(1,1)} & \ldots & Q^{(1,J)} \\ \vdots & \ddots & \vdots \\ Q^{(J,1)} & \ldots & Q^{(J,J)} \end{bmatrix}.$$

Block (i, j) of Q for $i, j = 1, \ldots, J$ can be written as

$$Q^{(i,j)} = \begin{cases} \sum_{t \in \mathcal{T}^{(i,j)}} Q_t^{(i,j)} + Q_D^{(i)} & \text{if } i = j, \\ \sum_{t \in \mathcal{T}^{(i,j)}} Q_t^{(i,j)} & \text{otherwise,} \end{cases}$$

where

$$Q_t^{(i,j)} = \alpha_t \bigotimes_{h=1}^{H} Q_{t,h}^{(i,j)}, \quad Q_D^{(i)} = -\sum_{j=1}^{J} \sum_{t \in \mathcal{T}^{(i,j)}} \operatorname{diag}(Q_t^{(i,j)} \mathbf{e}),$$

α_t is the rate associated with transition t, $\mathcal{T}^{(i,j)}$ is the set of transitions in block (i, j), and $Q_{t,h}^{(i,j)}$ is the submatrix of the transition matrix $Q_{t,h}$ whose row and column state spaces are $\mathcal{S}_h^{(i)}$ and $\mathcal{S}_h^{(j)}$, respectively. The advantage of partitioning the reachable state space is the elimination of unreachable states from the set of rows and columns of the generator matrix. Hence, unnecessary flops due to unreachable states are avoided. In passing to the shuffle algorithm, we also note that the rate of a Kronecker product term, α_t, can be eliminated by scaling one factor in the term with that rate.

2.1. Shuffle algorithm. In numerical methods used for analyzing Kronecker based MCs, vectors are multiplied with Kronecker product terms. Now, let $X_h \in \mathbb{R}^{r_h \times c_h}$ for $h = 1, \ldots, H$ be the rectangular factors in a given Kronecker product term. Then the shuffle algorithm is based on the identity

$$\bigotimes_{h=1}^{H} X_h = \prod_{h=1}^{H} (I_{\prod_{f=1}^{h-1} r_f} \otimes X_h \otimes I_{\prod_{f=h+1}^{H} c_f}),$$

which is due to compatibility of the Kronecker product with matrix multiplication [20]. Besides, $I_{\prod_{f=1}^{h-1} r_f} \otimes X_h \otimes I_{\prod_{f=h+1}^{H} c_f}$ is an identity matrix if $X_h = I_{r_h}$. Hence, the left-multiplication of $\mathbf{p} \in \mathbb{R}^{1 \times \prod_{h=1}^{H} r_h}$ with $X = \bigotimes_{h=1}^{H} X_h$ can be accomplished as in Algorithm 1. Note that this results in an output vector whose length ranges from $c_1 \prod_{h=2}^{H} r_h$ to $\prod_{h=1}^{H} c_h$ during the course of the multiplication.

Input: Vector: $\mathbf{p}$
Rectangular Kronecker product factors: $X_1, \ldots, X_H$
Output: Vector: $\mathbf{q} = \mathbf{p} \bigotimes_{h=1}^{H} X_h$:

```
1: function SHUFFLE(p, X_1, ..., X_H, q)
2:     Copy p to q;
3:     i_left = 1; i_right = ∏_{h=2}^{H} r_h; r_{H+1} = 1;
4:     for all h = 1 to H do
5:         if X_h ≠ I_{r_h} then
6:             base_i = 0; base_j = 0;
7:             for all i_l = 0, ..., i_left − 1 do
8:                 for all i_r = 0, ..., i_right − 1 do
9:                     index_i = base_i + i_r;
10:                    for all row = 0, ..., r_h − 1 do
11:                        z(row) = q(index_i); index_i = index_i + i_right;
12:                    end for
13:                    z' = z X_h;
14:                    index_j = base_j + i_r;
15:                    for all col = 0, ..., c_h − 1 do
16:                        q'(index_j) = z'(col); index_j = index_j + i_right;
17:                    end for
18:                end for
19:                base_i = base_i + r_h i_right; base_j = base_j + c_h i_right;
20:            end for
21:            Copy q' to q
22:        end if
23:        i_left = i_left c_h; i_right = i_right / r_{h+1};
24:    end for
25: end function
```

ALGORITHM 1. SHUFFLE ALGORITHM FOR RECTANGULAR FACTORS.

Now, observe that the only flops executed in Algorithm 1 are in line 13, where the temporary vector $\mathbf{z}$ is multiplied with the nonidentity matrix X_h. For a fixed value of h, this line is executed $\prod_{f=1}^{h-1} c_f \prod_{f=h+1}^{H} r_f$ times, and the cost of vector multiplication

with $\bigotimes_{h=1}^{H} X_h$ using the shuffle algorithm amounts to

$$2\sum_{h\in\mathcal{H}} nnz(X_h)\prod_{f=1}^{h-1} c_f \prod_{f=h+1}^{H} r_f \tag{2.1}$$

flops, where

$$\mathcal{H} = \{h \in \{1,\dots,H\} \mid X_h \neq I_{r_h}\}.$$

When $|\mathcal{H}| = 1$, we have that $r_h = c_h = nnz(X_h)$ for $h \notin \mathcal{H}$ and $h = 1,\dots,H$, and therefore, the number of flops executed by the shuffle algorithm is $2nnz(X)$

Algorithm 1 requires four vectors: $\mathbf{q}$, $\mathbf{q}'$, $\mathbf{z}$, and $\mathbf{z}'$ in addition to the input vector. The vectors $\mathbf{q}$ and $\mathbf{q}'$ are the floating-point vectors storing the intermediate results; hence, each of them needs to be as long as $\max_{h\in\mathcal{H}}(\prod_{f=1}^{h} c_f \prod_{f=h+1}^{H} r_f)$. When the input vector needs to be multiplied with sums of Kronecker product terms, the output vector cannot be used to keep intermediate results. The floating-point vectors $\mathbf{z}$ and $\mathbf{z}'$ need to be at least as long as $\max_h(r_h)$ and $\max_h(c_h)$, respectively.

The next example is given to show how vector-Kronecker product multiplication works in the presence of rectangular factors when Algorithm 1 is used.

Example 2.2. Consider the multiplication of the vector

$$\mathbf{p} = [a_0 \;\; a_1 \;\; a_2 \;\; a_3 \;\; a_4 \;\; a_5 \;\; a_6 \;\; a_7 \;\; a_8 \;\; a_9 \;\; a_{10} \;\; a_{11} \;\; a_{12} \;\; a_{13} \;\; a_{14} \;\; a_{15} \;\; a_{16} \;\; a_{17}]$$

with the Kronecker product term $X = X_1 \otimes X_2 \otimes X_3$, where

$$X_1 = \begin{bmatrix} 3 & \\ 2 & \\ & \end{bmatrix}_{3\times 2}, \quad X_2 = \begin{bmatrix} & 2 \\ & \end{bmatrix}_{3\times 2}, \quad \text{and} \quad X_3 = \begin{bmatrix} 5 & \\ 1 & 3 \end{bmatrix}_{2\times 3}.$$

In Algorithm 1, the outer loop in line 4 is executed three times for $h = 1, 2, 3$. At the beginning of the algorithm, $\mathbf{p}$ is copied to $\mathbf{q}$ and $\mathbf{q}' = \mathbf{q}(X_1 \otimes I_6)$ is computed for $h = 1$. In this turn, $nnz(X_1) = 2$, $i_{left} = 1$, and $i_{right} = 6$. Therefore, the middle loop in line 7 and the inner loop in line 8 are executed once and six times, respectively. Then $\mathbf{q}(X_1 \otimes I_6)$ is computed as

$$\mathbf{q}' = [3a_0 + 2a_6 \;\; 3a_1 + 2a_7 \;\; 3a_2 + 2a_8 \;\; 3a_3 + 2a_9 \;\; 3a_4 + 2a_{10} \;\; 3a_5 + 2a_{11} \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0]$$

in 24 flops and $\mathbf{q}'$ is copied to $\mathbf{q}$ at the end of the first turn. In the second turn, $\mathbf{q}' = \mathbf{q}(I_2 \otimes X_2 \otimes I_2)$ is computed for $h = 2$. In this turn, $nnz(X_2) = 1$, $i_{left} = 2$, and $i_{right} = 2$. Therefore, the middle loop and the inner loop are each executed twice. Then $\mathbf{q}(I_2 \otimes X_2 \otimes I_2)$ is computed as

$$\mathbf{q}' = [0 \;\; 0 \;\; 6a_2 + 4a_8 \;\; 6a_3 + 4a_9 \;\; 0 \;\; 0 \;\; 0 \;\; 0]$$

in 8 flops and $\mathbf{q}'$ is copied to $\mathbf{q}$ at the end of the second turn. In the last turn, $\mathbf{q}' = \mathbf{q}(I_4 \otimes X_3)$ is computed for $h = 3$. In this turn, $nnz(X_3) = 3$, $i_{left} = 4$, and $i_{right} = 1$. Therefore, the middle loop and the inner loop are executed four times and once, respectively. Then $\mathbf{q}(I_4 \otimes X_3)$ is computed as

$$\mathbf{q}' = [0 \;\; 0 \;\; 0 \;\; 0 \;\; 30a_2 + 6a_3 + 20a_8 + 4a_9 \;\; 18a_3 + 12a_9 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0]$$

in 24 flops, and $\mathbf{q}'$ is copied back to $\mathbf{q}$ at the end of the last turn. Hence, Algorithm 1's computation of $\mathbf{q} = \mathbf{p}X$ takes altogether 56 flops.

2.2. Pot-RwCl algorithm. A second approach for computing $\mathbf{p}\bigotimes_{h=1}^{H} X_h$ is to generate nonzero elements of the Kronecker product term and multiply them with corresponding elements of the vector. A nonzero element of the Kronecker product term is obtained as a result of the multiplication of H nonzeros each coming from a different factor. The Pot-RwCl algorithm proposed in [13] exploits common multipliers of nonzero elements of Kronecker product terms so that multiplications with the same values are avoided. Hence, $\mathbf{p}\bigotimes_{h=1}^{H} X_h$ can be computed recursively by calling Algorithm 2 with the initial parameters $h = 1$, $i = 0$, $j = 0$, and $v = 1$.

Input: Vector: $\mathbf{p}$
Rectangular Kronecker product factors: $X_1, \ldots, X_H$
Subsystem: h
Row index obtained from submatrices $1, \ldots, h-1$: i
Column index obtained from submatrices $1, \ldots, h-1$: j
Nonzero obtained from submatrices $1, \ldots, h-1$: v
Output: Vector: $\mathbf{q}$ such that $\mathbf{q}(\mathcal{J}') = \mathbf{q}(\mathcal{J}') + \mathbf{p}(\mathcal{I}')\ (v \bigotimes_{f=h}^{H} X_f)$, where $\mathcal{I}' = \{i, \ldots, i + \prod_{f=h}^{H} r_f - 1\}$ and $\mathcal{J}' = \{j, \ldots, j + \prod_{f=h}^{H} c_f - 1\}$

```
1:  function Pot-RwCl(p, X_1, ..., X_H, h, i, j, v, q)
2:      for all (i_h, j_h) such that X_h(i_h, j_h) > 0 do
3:          i' ← i + i_h ∏_{f=h+1} r_f; j' ← j + j_h ∏_{f=h+1} c_f;
4:          if X_h ≠ I_{r_h} then
5:              v' ← v X_h(i_h, j_h);
6:          end if
7:          if h < H then
8:              Pot-RwCl(p, X_1, ..., X_H, h + 1, i', j', v', q)
9:          else
10:             q(j') = q(j') + p(i') v';
11:         end if
12:     end for
13: end function
```

Algorithm 2. Pot-RwCl algorithm for rectangular factors.

Observe that the only flops executed in Algorithm 2 are in lines 5 and 10. In line 5, multiplication of nonzeros from subsystems 1 through h are performed. For a fixed value of h, the function is called $\prod_{f=1}^{h-1} nnz(X_f)$ times. Hence, the number of flops executed in line 5 is $\prod_{f=1}^{h} nnz(X_f)$ if $X_h \neq I_{r_h}$. On the other hand, line 10 is executed for each nonzero element of the Kronecker product of factors. Therefore, the cost of vector-Kronecker product multiplication using the Pot-RwCl algorithm amounts to

$$\sum_{h \in \mathcal{H}} \prod_{f=1}^{h} nnz(X_f) + 2\ nnz(X) \tag{2.2}$$

flops. Since Algorithm 2 does not have any intermediate vector computations, it does not require any additional floating-point vectors.

Next, we demonstrate how Pot-RwCl works using our running example.

Example 2.2 (continued). The matrix $X = X_1 \otimes X_2 \otimes X_3$ includes six nonzero elements which can be written as

$$X(2,4) = X_1(0,0)X_2(1,1)X_3(0,1),\ X(3,4) = X_1(0,0)X_2(1,1)X_3(1,1),$$

$$X(3,5) = X_1(0,0)X_2(1,1)X_3(1,2),\ X(8,4) = X_1(1,0)X_2(1,1)X_3(0,1),$$

$$X(9,4) = X_1(1,0)X_2(1,1)X_3(1,1),\ X(9,5) = X_1(1,0)X_2(1,1)X_3(1,2),$$

where $X_1(0,0) = 3$, $X_1(1,0) = 2$, $X_2(1,1) = 2$, $X_3(0,1) = 5$, $X_3(1,1) = 1$, and $X_3(1,2) = 3$. Initially the function is called for the values $h = 1$, $i = 0$, $j = 0$, and $v = 1$. Since X_1 includes two nonzeros, the loop in line 2 is executed twice. In the first turn, the function in line 8 is called for the values $h = 2$, $i' = 0$, $j' = 0$, and $v' = 3$. Then the loop in line 2 is executed once for the single nonzero element of X_2 and the function in line 8 is called for the values $h = 3$, $i' = 2$, $j' = 3$, and $v' = 6$. Since X_3 includes three nonzero elements, the loop in line 2 is executed three times. In these turns, $X(2,4) = 30$, $X(3,4) = 6$, and $X(3,5) = 18$ are obtained and $\mathbf{q}$ is computed as

$$\mathbf{q} = [0\ \ 0\ \ 0\ \ 0\ \ 30a_2 + 6a_3\ \ 18a_3\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0]$$

in 11 flops. The second turn of the loop in line 2 of the function is executed for $X_1(1,0)$ and the function in line 8 is called for the values $h = 2$, $i' = 6$, $j' = 0$, and $v' = 2$. Then the loop in line 2 is executed once for the single nonzero element of X_2 and the function in line 8 is called for the values $h = 3$, $i' = 8$, $j' = 3$, and $v' = 4$. Then $X(8,4) = 20$, $X(9,4) = 4$, and $X(9,5) = 12$ are obtained and $\mathbf{q}$ is computed as

$$\mathbf{q} = [0\ \ 0\ \ 0\ \ 0\ \ 30a_2 + 6a_3 + 20a_8 + 4a_9\ \ 18a_3 + 12a_9\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0]$$

in 22 flops.

The next section introduces the proposed modification on the shuffle algorithm and analyzes its merit through a lemma.

3. Modified shuffle algorithm. In the shuffle algorithm, the number of flops executed due to a given factor does not depend on the nonzero structure of the other factors in the Kronecker product. However, when other factors include zero rows or columns, some vector elements end up being computed even though they would evaluate to zero at the end. A speculative example is the multiplication of a vector with a Kronecker product term including a zero factor. The shuffle algorithm executes possibly a large number of flops even though the result vector evaluates to zero at the end.

In this section, we present the proposed modification to the shuffle algorithm so that only relevant elements of the vector are multiplied with the nonzeros of factors and unnecessary flops are avoided. The following lemma provides the identity on which the modification is based.

LEMMA 3.1. *Let $P_{(u,\mathcal{U})} \in \{0,1\}^{u \times |\mathcal{U}|}$ be the matrix given elementwise as*

$$P_{(u,\mathcal{U})}(i,j) \begin{cases} 1 & \text{if } j = f^{(\mathcal{U})}(i) \text{ and } i \in \mathcal{U}, \\ 0 & \text{otherwise,} \end{cases}$$

where u is a finite positive integer, $\mathcal{U} \subseteq \{0,\dots,u-1\}$ is a nonempty set, and

$$f^{(\mathcal{U})}(i) = |\{j \in \mathcal{U} \mid j < i\}| \quad \text{for } i \in \mathcal{U}.$$

Besides, let $X_h \in \mathbb{R}^{r_h \times c_h}$, $\mathcal{R}_h \subseteq \{0,\dots,r_h-1\}$ and $\mathcal{C}_h \subseteq \{0,\dots,c_h-1\}$ denote the sets of rows and columns that include at least one nonzero in X_h for $h = 1,\dots,H$. Then

$$\bigotimes_{h=1}^{H} X_h = \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)} \bigotimes_{h=1}^{H} \hat{X}_h \bigotimes_{h=1}^{H} P^T_{(c_h,\mathcal{C}_h)}, \tag{3.1}$$

where

$$\hat{X}_h = P^T_{(r_h,\mathcal{R}_h)}\ X_h\ P_{(c_h,\mathcal{C}_h)} \quad \textit{for } h = 1,\ldots,H.$$

Proof. Observe that $\bar{P}_{(u,\mathcal{U})} = P_{(u,\mathcal{U})}P^T_{(u,\mathcal{U})}$ is a $(u \times u)$ diagonal matrix such that

$$\bar{P}_{(u,\mathcal{U})}(i,j) \begin{cases} 1 & \text{if } \ i = j \text{ and } i \in \mathcal{U} \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } i,j \in \{0,\ldots,u-1\}.$$

Then

$$X_h = \bar{P}_{(r_h,\mathcal{R}_h)}\ X_h \ \text{ and } \ X_h = X_h\ \bar{P}_{(c_h,\mathcal{C}_h)}$$

hold for $h = 1,\ldots,H$, from which it follows that

$$\begin{aligned}
\bigotimes_{h=1}^{H} X_h &= \bigotimes_{h=1}^{H} \bar{P}_{(r_h,\mathcal{R}_h)}\ X_h\ \bar{P}_{(c_h,\mathcal{C}_h)} \\
&= \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)}\ P^T_{(r_h,\mathcal{R}_h)}\ X_h\ P_{(c_h,\mathcal{C}_h)}\ P^T_{(c_h,\mathcal{C}_h)} \\
&= \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)}\ \hat{X}_h\ P^T_{(c_h,\mathcal{C}_h)} \\
&= \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)}\ \bigotimes_{h=1}^{H} \hat{X}_h\ \bigotimes_{h=1}^{H} P^T_{(c_h,\mathcal{C}_h)}
\end{aligned}$$

by the compatibility of Kronecker product with matrix multiplication. ∎

The modified shuffle algorithm is based on the next corollary.

COROLLARY 3.2. *Let* $\mathbf{q} = \mathbf{p}\bigotimes_{h=1}^{H} X_h$, *and let* $\hat{X}_h$, $\mathcal{R}_h$, *and* $\mathcal{C}_h$ *be defined as in Lemma* 3.1 *for* $h = 1,\ldots,H$. *Then*

$$\mathbf{q}(\times_{h=1}^{H}\mathcal{C}_h) = \mathbf{p}(\times_{h=1}^{H}\mathcal{R}_h)\ \bigotimes_{h=1}^{H} \hat{X}_h.$$

Proof. Using (3.1), $\mathbf{q} = \mathbf{p}\bigotimes_{h=1}^{H} X_h$ can be written as

$$\mathbf{q} = \mathbf{p}\ \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)}\ \bigotimes_{h=1}^{H} \hat{X}_h\ \bigotimes_{h=1}^{H} P^T_{(c_h,\mathcal{C}_h)}.$$

Then the result follows from

$$\mathbf{p}(\times_{h=1}^{H}\mathcal{R}_h) = \mathbf{p}\ \bigotimes_{h=1}^{H} P_{(r_h,\mathcal{R}_h)}, \quad \mathbf{q}(\times_{h=1}^{H}\mathcal{C}_h) = \mathbf{q}\ \bigotimes_{h=1}^{H} P_{(c_h,\mathcal{C}_h)},$$

and $P^T_{(c_h,\mathcal{C}_h)}P_{(c_h,\mathcal{C}_h)} = I_{|\mathcal{C}_h|}$ for $h = 1,\ldots,H$. ∎

In Algorithm 3, flops are executed only while multiplying the vector $\hat{\mathbf{p}}$ with $\otimes_{h=1}^{H}\hat{X}_h$. Hence, the cost of vector-Kronecker product multiplication using the modified shuffle algorithm amounts to

$$2\ \sum_{h\in\mathcal{H}} nnz(X_h) \prod_{f=1}^{h-1} |\mathcal{C}_f| \prod_{f=h+1}^{H} |\mathcal{R}_f| \tag{3.2}$$

Input: Vector: $\mathbf{p}$
Rectangular Kronecker product factors: $X_1, \ldots, X_H$
Output: Vector: $\mathbf{q} = \mathbf{p} \bigotimes_{h=1}^{H} X_h$:
1: **function** ModifiedShuffle($\mathbf{p}, X_1, \ldots, X_H, \mathbf{q}$)
2: **for all** $h = 1$ to H **do**
3: $\mathcal{R}_h = \emptyset$; $\mathcal{C}_h = \emptyset$;
4: **for all** (i_h, j_h) such that $X_h(i_h, j_h) > 0$ **do**
5: $\mathcal{R}_h \leftarrow \mathcal{R}_h \cup \{i_h\}$; $\mathcal{C}_h \leftarrow \mathcal{C}_h \cup \{j_h\}$;
6: **end for**
7: Copy $X_h(\mathcal{R}_h, \mathcal{C}_h)$ to $\hat{X}_h$;
8: **end for**
9: Copy $\mathbf{p}(\times_{h=1}^{H} \mathcal{R}_h)$ to $\hat{\mathbf{p}}$;
10: Shuffle($\hat{\mathbf{p}}, \hat{X}_1, \ldots, \hat{X}_H, \hat{\mathbf{q}}$)
11: Copy $\hat{\mathbf{q}}$ to $\mathbf{q}(\times_{h=1}^{H} \mathcal{C}_h)$;
12: **end function**

Algorithm 3. Modified shuffle algorithm for rectangular factors.

flops since $nnz(\hat{X}_h) = nnz(X_h)$ for $h = 1, \ldots, H$. Note that the cost of Algorithm 3 in (3.2) never exceeds the cost of Algorithm 1 in (2.1) and is bounded above by $(2\,|\mathcal{H}|\;nnz(X))$ flops since $|\mathcal{R}_h| \le nnz(X_h)$ and $|\mathcal{C}_h| \le nnz(X_h)$ for $h = 1, \ldots, H$.

Regarding memory requirements, each of the vectors $\mathbf{q}$ and $\mathbf{q}'$ in Algorithm 3 needs to be as long as $\max_{h\in\mathcal{H}}(\prod_{f=1}^{h} |\mathcal{C}_f| \prod_{f=h+1}^{H} |\mathcal{R}_h|)$, whereas the floating-point vectors $\mathbf{z}$ and $\mathbf{z}'$ need to be at least as long as $\max_h(|\mathcal{R}_h|)$ and $\max_h(|\mathcal{C}_h|)$, respectively.

We use our running example to show how the modified shuffle algorithm works.

Example 2.2 (continued). The sets of rows and columns of the factors X_1, X_2, and X_3 including at least one nonzero value are given by Algorithm 3 as

$$\mathcal{R}_1 = \{0,1\},\ \mathcal{C}_1 = \{0\},\ \mathcal{R}_2 = \{1\},\ \mathcal{C}_2 = \{1\},\ \mathcal{R}_3 = \{0,1\},\ \text{and } \mathcal{C}_3 = \{1,2\}.$$

Then

$$\times_{h=1}^{3}\mathcal{R}_h = \{(0,1,0),(0,1,1),(1,1,0),(1,1,1)\},\ \ \times_{h=1}^{3}\mathcal{C}_h = \{(0,1,1),(0,1,2)\},$$

$$P_{(r_1,\mathcal{R}_1)} = \begin{bmatrix} 1 & \\ & 1 \\ & \end{bmatrix}_{3\times 2},\ P_{(c_1,\mathcal{C}_1)} = \begin{bmatrix} 1 \\ \end{bmatrix}_{2\times 1},\ P_{(r_2,\mathcal{R}_2)} = \begin{bmatrix} 1 \\ \\ \end{bmatrix}_{3\times 1},$$

$$P_{(c_2,\mathcal{C}_2)} = \begin{bmatrix} 1 \\ \end{bmatrix}_{2\times 1},\ P_{(r_3,\mathcal{R}_3)} = I_2,\ P_{(c_3,\mathcal{C}_3)} = \begin{bmatrix} 1 & \\ & 1 \\ \end{bmatrix}_{3\times 2},$$

$$\bigotimes_{h=1}^{3} P_{(r_h,\mathcal{R}_h)} = [\mathbf{e}_2\ \ \mathbf{e}_3\ \ \mathbf{e}_8\ \ \mathbf{e}_9]_{18\times 4},\ \text{ and }\ \bigotimes_{h=1}^{3} P_{(c_h,\mathcal{C}_h)} = [\mathbf{e}_4\ \ \mathbf{e}_5]_{12\times 2}.$$

Hence,

$$\begin{aligned}\hat{\mathbf{p}} = \mathbf{p} \bigotimes_{h=1}^{3} P_{(r_h,\mathcal{R}_h)} &= \mathbf{p}(\times_{h=1}^{3}\mathcal{R}_h) \\ &= [a_2\ \ a_3\ \ a_8\ \ a_9].\end{aligned}$$

Then $\hat{\mathbf{q}} = \hat{\mathbf{p}} \ \otimes_{h=1}^{H} \hat{X}_h$ is computed using the shuffle algorithm, where

$$\hat{X}_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \quad \hat{X}_2 = \begin{bmatrix} 2 \end{bmatrix}, \quad \text{and} \quad \hat{X}_3 = \begin{bmatrix} 5 & \\ 1 & 3 \end{bmatrix}.$$

In Algorithm 1, the outer loop in line 4 is executed three times for $h = 1, 2, 3$. At the beginning of the algorithm, $\hat{\mathbf{p}}$ is copied to $\mathbf{q}$ and $\mathbf{q}' = \mathbf{q}\ (\hat{X}_1 \otimes I_2)$ is computed for $h = 1$. In this turn, $nnz(\hat{X}_1) = 2$, $i_{left} = 1$, and $i_{right} = 2$. Therefore, the middle loop in line 7 and the inner loop in line 8 are executed once and twice, respectively. Then $\mathbf{q}\ (\hat{X}_1 \otimes I_2)$ is computed as

$$\mathbf{q}' = [3a_2 + 2a_8 \quad 3a_3 + 2a_9]$$

in 8 flops and $\mathbf{q}'$ is copied to $\mathbf{q}$ at the end of the first turn. In the second turn, $\mathbf{q}' = \mathbf{q}\ (I_1 \otimes \hat{X}_2 \otimes I_2)$ is computed for $h = 2$. In this turn, $nnz(\hat{X}_2) = 1$, $i_{left} = 1$, and $i_{right} = 2$. Therefore, the middle loop and the inner loop are executed once and twice, respectively. Then $\mathbf{q}\ (I_1 \otimes \hat{X}_2 \otimes I_2)$ is computed as

$$\mathbf{q}' = [6a_2 + 4a_8 \quad 6a_3 + 4a_9]$$

in 4 flops and $\mathbf{q}'$ is copied to $\mathbf{q}$ at the end of the second turn. In the last turn, $\mathbf{q}' = \mathbf{q}\ (I_1 \otimes \hat{X}_3)$ is computed for $h = 3$. In this turn, $nnz(X_3) = 3$, $i_{left} = 1$, and $i_{right} = 1$. Therefore, the middle loop and the inner loop are each executed once. Then $\mathbf{q}\ (I_1 \otimes \hat{X}_3)$ is computed as

$$\mathbf{q}' = [30a_2 + 6a_3 + 20a_8 + 4a_9 \quad 18a_3 + 12a_9]$$

in 6 flops, and $\mathbf{q}'$ is copied to $\hat{\mathbf{q}}$ at the end of the last turn. Then the elements of $\hat{\mathbf{q}}$ are copied back to $\mathbf{q}(\times_{h=1}^{3} \mathcal{C}_h)$, that is,

$$\mathbf{q} = [0 \ \ 0 \ \ 0 \ \ 0 \ \ 30a_2 + 6a_3 + 20a_8 + 4a_9 \ \ 18a_3 + 12a_9 \ \ 0 \ \ 0 \ \ 0 \ \ 0 \ \ 0 \ \ 0]$$

by the equation

$$\hat{\mathbf{q}} = \mathbf{q} \bigotimes_{h=1}^{3} P_{(c_h, \mathcal{C}_h)} = \mathbf{q}(\times_{h=1}^{3} \mathcal{C}_h).$$

Hence, Algorithm 3's computation of $\mathbf{q} = \mathbf{p}\hat{X}$ takes altogether 18 flops. This number is smaller than the 56 flops and 22 flops that the shuffle and Pot-RwCl algorithms, respectively, take.

We remark that the proposed improvement will also be useful when the factors are relatively dense as long as some of them include zero rows or columns. In the next section, we discuss implementation issues associated with the shuffle, Pot-RwCl, and modified shuffle algorithms.

4. Implementation issues. We considered two benchmarks in this paper. The first one is the implementation of the shuffle algorithm in the Nsolve package of the Abstract Petri Net Notation toolbox [2, 4]. The pseudocode of the Nsolve implementation of the loop in line 4 of Algorithm 1 is given in [7]. Identity matrices are not stored as discussed in [7] and the multiplication is not performed for identity factors. In addition, we modified the implementation so that each matrix that is a multiple of an identity matrix is not stored either. In this implementation, each nonzero of the

factor X_h is multiplied with the elements of the input vector and added to the appropriate elements of the output vector so that the vectors $\mathbf{z}$ and $\mathbf{z}'$ are not used and stored in the multiplication. Since we consider models with more than one Kronecker product term, the intermediate results cannot be stored in the output vector. The number of auxiliary vectors needed to store the intermediate results depends on the maximum of the number of nonidentity factors across all Kronecker product terms. If this number is one, then there is no intermediate result, and therefore no auxiliary vector is required. If the maximum number of nonidentity factors across all terms is two, then the allocation of one auxiliary vector is sufficient. Otherwise, two auxiliary vectors need to be allocated to store the intermediate results. However, it should be noted that at least two auxiliary vectors each of length $\max_i |\mathcal{S}^{(i)}|$ are required in numerical solvers. This is the case for all algorithms discussed in this paper when they are used as kernels in numerical solvers. In the Nsolve implementation of the shuffle algorithm, the elements of the output vector are multiplied by the rate α_t associated with the Kronecker product term and added to the output vector after the multiplication of the input vector with the Kronecker product term. The cost of multiplying the output vector with α_t becomes excessive, especially when the factors are sparse. In order to avoid this cost, we multiplied α_t with the nonzero elements of the first nonidentity factor and stored the modified factor. If all factors are identity matrices, the term rate is kept separately and the multiplication is obtained by multiplying the rate with the input vector.

As the second benchmark, we implemented the Pot-RwCl algorithm for rectangular factors. The pseudocode of the algorithm given in [13] for square factors includes H nested loops, but therein it is also stated that a recursive implementation is required since H is model dependent. In order to make a fair comparison between the algorithms, we provided a nonrecursive implementation with dynamically allocated arrays of size H as suggested in [13]. In the Pot-RwCl algorithm, row and column indices of the nonzero elements of Kronecker product terms need to be computed. Hence, its indexing and addressing overhead is larger than that of the shuffle algorithm. However, Pot-RwCl uses the cache more efficiently since nonzero elements of the Kronecker product of the factors are obtained consecutively and the corresponding elements of the input and output vectors tend to be closer to each other. In the shuffle algorithm, the cache is not accessed as efficiently as nonzero generation that takes place in the Pot-RwCl algoritm when the value of index h is large. Furthermore, as stated in subsection 2.2, Pot-RwCl does not require any intermediate vector computation, implying it does not require any additional floating-point vectors (except the two auxiliary vectors each of length $\max_i |\mathcal{S}^{(i)}|$ when it is used as a kernel in numerical solvers). To cut down on indexing overhead in the implementation of line 8 in Algorithm 2, the input vector is multiplied with the common nonzero element when the remaining factors to be processed are identity matrices even if $h < H$.

In the implementation of the modified shuffle algorithm, we removed zero rows and columns of the nonidentity matrices and allocated two integer vectors to store the mapping between the row and column indices of the factor and the modified factor. This modification may avoid unnecessary flops only if the Kronecker product term includes more than one nonidentity factor. In addition, choosing elements of the input and output vectors yields indexing and addressing overhead in many cases. Therefore, we removed zero rows and columns of nonidentity factors in a Kronecker product term only if their counts are more than one. A naive implementation is to copy appropriate elements of the input vector to a temporary vector, multiply this temporary vector with the Kronecker product of the submatrices, and add the resulting vector to the

output vector. Instead, we modified the shuffle algorithm implementation so that appropriate elements of the input vector are chosen in the first turn of the outer loop in line 4 of Algorithm 1. Similarly, the elements of the resulting vector are added to the appropriate elements of the output vector in the last turn of the outer loop in line 4 of Algorithm 1. A recursive implementation seemed to be convenient in choosing relevant elements of input and output vectors. However, in order to enable a fair comparison between the algorithms, we implemented Algorithm 3 without recursion using arrays of size H as it is done for Pot-RwCl. We preferred the same implementation even for the multiplication of the vector with Kronecker product terms in which no rows and columns are removed from the factors. Implementations of the three algorithms as discussed here are available at [18].

5. Numerical results. We performed numerical experiments on an Intel Core2 Duo 2.4 GHz processor with 4 GB of main memory. We considered 15 Kronecker based MCs with given Cartesian product partitionings of their reachable state spaces. Six of the MCs correspond to systems of stochastic chemical kinetics that are models of a gene expression [32], a toggle switch [22], an exclusive switch [26], a metabolite synthesis with one enzyme [30], a metabolite synthesis with two enzymes [30], and a repressilator [25]. One of the MCs is a queueing network model of a call center having five subsystems; it is a parallel service system known as the N-model under the threshold routing control policy proposed in [5]. Further information regarding these seven models may be obtained from [3, 19]. The remaining eight MCs are models of the Courier protocol introduced in [34], the multiserver multiqueue (MSMQ) models discussed in [1], models of manufacturing systems having Kanban control [27], and a model of token based scheduling in a queueing network called qh_realcontrol [11]. These were used as benchmark problems before [8, 9, 10, 11].

Table 1 provides the properties associated with the Kronecker representation of the generator matrices of the MC models. The first column gives the name of the model, the second column gives the number of subsystems in the corresponding system, the third column gives the number of reachable state space partitions, the fourth column gives the number of Kronecker product terms in the representation, the fifth column gives the average number of nonzeros per row in the factors, the sixth column gives the maximum partition size of the reachable state space partitioning, the seventh column gives the number of reachable states, and the last column gives the number of nonzero off-diagonal elements in Q underlying the respective MC. Note that for practical reasons, in almost all cases the diagonal of Q is stored explicitly in Kronecker based MCs; hence, for a fairer memorywise comparison, we also do not account for that in the flat representation. Otherwise, the number of nonzeros in Q for each model may be found by summing the value in the last two columns of Table 1. The values of nonzeros in Q is immaterial for this work; hence, we do not provide the real parameters of the corresponding models.

We considered models of different sizes. The reachable state space sizes of the models range from about 350,000 to about 3,000,000. All models except gene expression have at least three factors in their Kronecker products. The models coming from stochastic chemical physics, kanban_medium, and kanban_large do not have any unreachable states. Recall that all square factors is just a special case of rectangular factors, and by experimenting with these models we show that indeed there are savings that may be realized in the all square factors case as well. The maximum number of Kronecker product terms over all blocks of the generator matrix is 1,100 and appears in the N-model. It is also the N-model which has the largest number of partitions of

TABLE 1
Model properties

Model	H	prt	$term$	$dens$	$\max_i \lvert\mathcal{S}^{(i)}\rvert$	$\lvert\mathcal{S}\rvert$	$nnz(Q_{off})$
gene expression	2	1	4	1.00	1,002,001	1,002,001	4,003,000
toggle switch	3	1	8	0.87	1,085,764	1,085,764	5,418,400
exclusive switch	3	1	10	0.82	910,803	910,803	4,242,700
met. syn. one enz.	3	1	7	1.00	1,191,016	1,191,016	8,236,200
met. syn. two enz.	4	1	9	1.00	1,679,616	1,679,616	14,560,560
repressilator	4	1	12	0.90	1,191,016	1,191,016	8,764,080
N-model	5	204	1,100	0.96	39,601	489,930	2,334,358
courier_large	4	10	80	0.91	117,000	419,400	2,281,620
courier_huge	4	13	109	0.91	468,000	1,632,600	9,732,330
msmq_medium	5	15	100	0.98	26,136	358,560	2,135,160
msmq_large	5	35	250	0.99	107,653	2,945,880	19,894,875
kanban_medium	4	1	7	0.95	527,076	527,076	3,001,405
kanban_large	4	1	7	0.96	1,742,400	1,742,400	10,183,360
kanban_fail	4	8	68	0.93	291,600	2,302,911	14,313,663
qh_realcontrol	3	9	45	0.99	48,048	399,476	1,871,004

its reachable state space with 204 partitions. The average number of nonzeros per row in the nonidentity factors of Kronecker product terms of the models ranges from 0.82 (exclusive switch) to 1.00 (gene expression, metabolite synthesis with one and two enzymes). Hence, Kronecker product factors of the models are relatively sparse for the shuffle algorithm to be efficient. If we were to compute the generator matrices corresponding to the models, we would be having $nnz(Q_{off})$ nonzeros in their off-diagonal parts and we would be performing $2nnz(Q_{off})$ flops when we multiply a vector of length $|\mathcal{S}|$ with Q_{off}. Note that $2nnz(Q_{off})$ flops is therefore a lower bound for the case of relatively sparse Kronecker product factors, and we should be happy to see flop counts that are close to $2nnz(Q_{off})$ with vector-Kronecker product multiplication algorithms.

Results of numerical experiments with the models in Table 1 are reported in Table 2. For each model, we considered implementations of the three vector-Kronecker product algorithms as discussed in section 4. In the second column of the table, S, P, and MS respectively denote the shuffle, the Pot-RwCl, and the modified shuffle algorithms. Column *nnz* provides the total number of nonzeros in the explicitly stored matrices. Column *expl* provides the number of explicitly stored matrices. Column *mexpl* provides the maximum of the number of explicitly stored matrices in the Kronecker product terms. Column $E[expl]$ provides the expected number of explicitly stored matrices. Column *subm* is the number of matrices in which at least one row or one column is removed. The last three columns pertain to the performance of vector-Kronecker product multiplication. Column *aux* provides the total length of additional floating-point vectors used during multiplication. This number is determined by *mexpl* as discussed in section 4. Column *flops* provides the number of flops executed in the multiplication of a vector with the off-diagonal part of Q in Kronecker form. The number of flops is the sum of the costs discussed in sections 2 and 3 over the Kronecker product terms in the blocks. The last column gives the time of the multiplication of a vector with the off-diagonal part of Q in Kronecker form in milliseconds of CPU time averaged over 10,000 such multiplications. We remark that the time it takes to set the Kronecker representation of the infinitesimal generator matrix is negligible. The bold figures in the last three columns of Table 2 indicate the minimum values in those columns for the particular model.

TABLE 2
Numerical results

Model	Alg.	*nnz*	*expl*	*mexpl*	*E*[*expl*]	*subm*	*aux*	*flops*	Time
gene expression	S	5,000	5	2	1.25	0	1,002,001	10,010,000	60
	P	5,000	5	2	1.25	0	**0**	10,010,000	38
	MS	4,000	4	1	1.00	2	**0**	**8,006,000**	**32**
toggle switch	S	4,172	14	2	1.75	0	1,085,764	23,853,464	133
	P	4,172	14	2	1.75	0	**0**	15,173,600	101
	MS	2,080	4	1	0.50	12	**0**	**10,836,800**	**62**
exclusive switch	S	5,508	18	2	1.80	0	910,803	23,040,616	111
	P	5,508	18	2	1.80	0	**0**	12,427,800	111
	MS	2,200	4	1	0.40	16	**0**	**8,485,400**	**74**
met. syn. one enz.	S	1,051	10	2	1.43	0	1,191,016	23,618,072	126
	P	1,051	10	2	1.43	0	**0**	**20,034,316**	**71**
	MS	946	9	2	1.29	5	1,191,016	21,147,000	83
met. syn. two enz.	S	492	14	2	1.56	0	1,679,616	45,909,504	323
	P	492	14	2	1.56	0	**0**	**34,114,642**	**116**
	MS	422	12	2	1.33	8	1,679,616	38,646,720	146
repressilator	S	660	21	2	1.75	0	1,191,016	38,764,200	321
	P	660	21	2	1.75	0	**0**	23,382,112	135
	MS	312	6	1	0.50	18	**0**	**17,528,160**	**77**
N-model	S	30,614	682	2	0.62	0	39,601	6,437,428	14
	P	30,614	682	2	0.62	0	**0**	5,199,918	12
	MS	26,104	594	1	0.54	88	**0**	**4,668,716**	**10**
courier_large	S	1,930	118	2	1.48	0	468,000	8,035,980	24
	P	1,930	118	2	1.48	0	**0**	5,293,496	14
	MS	1,517	57	1	0.71	76	**0**	**4,563,240**	**12**
courier_huge	S	4,540	162	2	1.49	0	1,404,000	38,095,740	241
	P	4,540	162	2	1.49	0	**0**	23,025,542	103
	MS	4,291	98	2	0.90	106	468,000	**20,073,660**	**80**
msmq_medium	S	985	125	2	1.25	0	47,916	5,193,936	**9**
	P	985	125	2	1.25	0	**0**	4,852,280	13
	MS	745	85	1	0.85	50	**0**	**4,270,320**	11
msmq_large	S	3,535	325	2	1.30	0	199,927	49,927,654	**109**
	P	3,535	325	2	1.30	0	**0**	45,177,891	130
	MS	3,115	265	2	1.06	150	107,653	**42,097,454**	111
kanban_medium	S	370	10	2	1.43	0	527,076	9,104,040	55
	P	370	10	2	1.43	0	**0**	6,996,185	24
	MS	130	4	1	0.57	6	**0**	**6,002,810**	**20**
kanban_large	S	406	10	2	1.43	0	1,742,400	30,666,240	231
	P	406	10	2	1.43	0	**0**	23,610,383	85
	MS	148	4	1	0.57	6	**0**	**20,366,720**	**72**
kanban_fail	S	2,916	108	2	1.59	0	364,500	50,222,268	186
	P	2,916	108	2	1.59	0	**0**	**35,183,125**	**175**
	MS	2,895	93	2	1.37	80	291,600	38,558,718	182
qh_realcontrol	S	2,331	63	2	1.40	0	70,070	5,199,112	**12**
	P	2,331	63	2	1.40	0	**0**	**4,618,858**	15
	MS	2,331	63	2	1.40	32	48,048	4,866,972	15

For each model, number of flops in vector-Kronecker product multiplication decreases when the shuffle algorithm is modified. There are two reasons for this decrease. First, unnecessary flops in the loop in line 4, for some fixed h, in Algorithm 1 are avoided. Second, some factors become multiples of identity matrices once the removal of zero rows and columns take place, and the loop in line 4 of Algorithm 1 is avoided for such factors. The improvement in number of flops is smallest at 6% for qh_realcontrol, the only model where the number of explicitly stored matrices does not decrease after the modification. In the other models, the improvement in the number of flops ranges from 10% (metabolite syntesis with one enzyme) to 63% (exclusive switch) with an average of 31% over 15 models. In all models but six (metabolite syntesis with one and two enzymes, courier_huge, msmq_large, kanban_fail, and qh_realcontrol), the modified shuffle algorithm performs $2nnz(Q_{off})$ flops, which is the lower bound. We remark that in each of these nine models, there is no term with

more than one explicitly stored matrix (see the values in column *mexpl*). Moreover, in courier_huge and msmq_large, modified shuffle yields the minimum number of flops among the three algorithms. Hence, modified shuffle is a winner in terms of number of flops in 11 models. In the nine models where modified shuffle performs $2nnz(Q_{off})$ flops, it also does not require additional memory. Besides, the modification also decreases the memory allocated for auxiliary vectors over that of shuffle in some other models such as courier_huge, msmq_large, kanban_fail, and qh_realcontrol. As expected, the number of flops executed by the Pot-RwCl algorithm is not more than that of the shuffle algorithm since the factors of Kronecker product terms in all models are relatively sparse. Pot-RwCl yields a smaller number of flops than modified shuffle only in four models (metabolite synthesis with one and two enzymes, kanban_fail, and qh_realcontrol). It is the value of $E[expl]$, which is relatively larger than 1, that causes modified shuffle to suffer in these four models. The number of flops for Pot-RwCl and shuffle are the same in gene expression. The equality is due to the choice of integer model parameters. For N-model and msmq_large, the difference in number of flops between Pot-RwCl and modified shuffle is not larger than 10%. In the other models, modified shuffle yields improvements in number of flops ranging from 12% to 36% over Pot-RwCl. Note that it is not feasible to use Pot-RwCl in models having relatively denser factors for which shuffle starts becoming quite efficient. Therefore, the results on these 15 models are in some sense indicative of the best Pot-RwCl can do.

The improvement in the number of flops obtained with modified shuffle in (3.2) can be predetermined by comparing it with (2.1) and (2.2). This is also true for the improvement in memory, if there is any. But, the improvement in time depends on the particular factors in the Kronecker product terms. Besides the number of flops, the time that the algorithms take also depends on the overhead of indexing and addressing as well as the access pattern to the input and output vectors. Time per flop (i.e., Time/$flops$) seems to be a good measure to understand the overhead and cache usage of algorithms. Time per flop of shuffle and Pot-RwCl are almost the same for the N-model. Shuffle is better than Pot-RwCl in time per flop for exclusive switch, msmq_medium, msms_large, kanban_fail, and qh_realcontrol. Shuffle takes smaller time than Pot-RwCl for msmq_medium, msms_large, and qh_realcontrol. In the other models, Pot-RwCl is timewise better than shuffle. This indicates that the cost due to indexing, addressing, and access pattern is smaller than the gain obtained from the decrease in the number of flops. On the other hand, time per flop of modified shuffle is larger than that of shuffle in exclusive switch, msmq_medium, msms_large, kanban_fail, and qh_realcontrol. Except exclusive switch, the decrease in the number of flops does not compensate for the overhead in the implementation. In other models, modified shuffle yields an improvement of up to 76% (repressilator) in time over shuffle. Time per flops of modified shuffle and Pot-RwCl are close to each other since these algorithms are implemented in a similar manner as discussed in section 4. Modified shuffle has overhead due to the mapping between rows and columns of the original and modified factors. However, it requires a smaller number of index operations in some models. The effect of these together seems to cancel, and the difference between time per flop values of Pot-RwCl and modified shuffle ends up being relatively small, generally in favor of modified shuffle.

6. Conclusion. This work shows how performance of vector-Kronecker product multiplication with rectangular factors can be improved when the factors are relatively sparse. The proposed approach is based on modifying the shuffle algorithm to avoid floating-point operations that evaluate to zero during the course of multiplication

by omitting zero rows and columns in the factors of Kronecker products. In many cases, this modified algorithm requires a smaller number of flops (and sometimes the minimum that is possible) compared to the traditional shuffle algorithm and another algorithm that generates nonzeros of the Kronecker product matrix on the fly. In addition, the modification is likely to decrease the memory requirement over that of the shuffle algorithm, in some cases to the extent that no additional memory is needed.

Acknowledgment. We thank the anonymous referees for their constructive reports that led to an improved manuscript.

REFERENCES

[1] M. Ajmone Marsan, S. Donatelli, and F. Neri, *GSPN models of Markovian multiserver multiqueue system*, Perform. Eval., 11 (1990), pp. 227–240.
[2] APNN-Toolbox. http://www4.cs.uni-dortmund.de/APNN-TOOLBOX (2004).
[3] H. Baumann, T. Dayar, M. C. Orhan, and W. Sandmann, *On the numerical solution of Kronecker-based infinite level-dependent QBD processes*, Perform. Eval., 70 (2013), pp. 663–681.
[4] F. Bause, P. Buchholz, and P. Kemper, *A toolbox for functional and quantitative analysis of DEDS*, in Quantitative Evaluation of Computing and Communication Systems, R. Puigjaner, N. N. Savino, and B. Serra, eds., Lecture Notes in Comput. Sci. 1469, Springer-Verlag, New York, 1998, pp. 356–359.
[5] S. L. Bell and R. J. Williams, *Dynamic scheduling of a parallel server system in heavy traffic with complete resource pooling: asymptotic optimality of a threshold policy*, Ann. Appl. Probab., 11 (2001), pp. 608–649.
[6] A. Benoit, B. Plateau, and W. J. Stewart, *Memory-efficient Kronecker algorithms with applications to the modeling of parallel systems*, Future Generation Comput. Syst., 22 (2006) pp. 838–847.
[7] P. Buchholz, *A class of hierarchical queueing networks and their analysis*, Queueing Syst., 15 (1994), pp. 59–80.
[8] P. Buchholz, *Adaptive decomposition and approximation for the analysis of stochastic Petri nets*, Perform. Eval., 56 (2004), pp. 23–52.
[9] P. Buchholz and T. Dayar, *Block SOR for Kronecker structured Markovian representations*, Linear Algebra Appl., 386 (2004), pp. 83–109.
[10] P. Buchholz and T. Dayar, *Comparison of multilevel methods for Kronecker-based Markovian representations*, Computing, 73 (2004), pp. 349–371.
[11] P. Buchholz and T. Dayar, *Block SOR preconditioned projection methods for Kronecker structured Markovian representations*, SIAM J. Sci. Comput., 26 (2005), pp. 1289–1313.
[12] P. Buchholz and T. Dayar, *On the convergence of a class of multilevel methods for large, sparse Markov chains*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 1025–1049.
[13] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper, *Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models*, INFORMS J. Comput., 12 (2000), pp. 203–222.
[14] D. Cheng, H. Qi, and Y. Zhao, *An Introduction to Semi-tensor Product of Matrices and Its Applications*, World Scientific, Singapore, 2012.
[15] M. Davio, *Kronecker products and shuffle algebra*, IEEE Trans. Comput., C-30 (1981), pp. 116–125.
[16] T. Dayar, *Analyzing Markov Chains Using Kronecker Products: Theory and Applications*, Springer, New York, 2012.
[17] T. Dayar and M. C. Orhan, *Cartesian Product Partitioning of Multi-Dimensional Reachable State Spaces*, Technical report BU-CE-1303, Department of Computer Engineering, Bilkent University, Ankara, Turkey, 2013; available online from http://www.cs.bilkent.edu.tr/tech-reports/2013/BU-CE-1303.pdf.
[18] T. Dayar and M. C. Orhan, *Vector–Kronecker Product Multiplication software*, http://www.cs.bilkent.edu.tr/~tugrul/software.html (2015).
[19] T. Dayar, W. Sandmann, D. Spieler, and V. Wolf, *Infinite level-dependent QBD processes and matrix analytic solutions for stochastic chemical kinetics*, Adv. Appl. Probab., 43 (2011), pp. 1005–1026.

[20] P. Fernandes, B. Plateau, and W. J. Stewart, *Efficient descriptor-vector multiplications in stochastic automata networks*, J. ACM, 45 (1998), pp. 381–414.
[21] P. Fernandes, B. Plateau, and W. J. Stewart, *Optimizing tensor product computations in stochastic automata networks*, RAIRO Rech. Opér., 32 (1998), pp. 325–351.
[22] T. S. Gardner, C. R. Cantor, and J. J. Collins, *Construction of a genetic toggle switch in Escherichia coli*, Nature, 403 (2000), pp. 339–342.
[23] I. Gurvich, M. Armony, and A. Mandelbaum, *Service-level differentiation in call centers with fully flexible servers*, Management Sci., 54 (2008), pp. 279–294.
[24] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems*, Cambridge University Press, New York, 2013.
[25] A. Loinger and O. Biham, *Stochastic simulations of the repressilator circuit*, Phys. Rev. E, 76 (2007), 051917.
[26] A. Loinger, A. Lipshtat, N. Q. Balaban, and O. Biham, *Stochastic simulations of genetic switch systems*, Phys. Rev. E, 75 (2007), 021904.
[27] D. Mitra and I. Mitrani, *Analysis of a Kanban discipline for cell coordination in production lines,* II*: Stochastic demands*, Oper. Res., 39 (1991), pp. 807–823.
[28] B. Plateau, *On the stochastic structure of parallelism and synchronization models for distributed algorithms*, Perform. Eval. Rev., 13 (1985), pp. 147–154.
[29] B. Plateau and J.-M. Fourneau, *A methodology for solving Markov models of parallel systems*, J. Parallel Distrib. Comput., 12 (1991), pp. 370–387.
[30] P. L. Sjöberg, P. Lötstedt, and J. Elf, *Fokker-Planck approximation of the master equation in molecular biolog*, Comput. Vis. Sci., 12 (2009), pp. 37–50.
[31] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, Princeton, NJ, 1994.
[32] M. Thattai and A. van Oudenaarden, *Intrinsic noise in gene regulatory networks*, Proc. Natl. Acad. Sci., 98 (2001), pp. 8614–8619.
[33] C. F. Van Loan, *The ubiquitous Kronecker product*, J. Comput. Appl. Math., 123 (2000), pp. 85–100.
[34] C. M. Woodside and Y. Li, *Performance Petri net analysis of communications protocol software by delay equivalent aggregation*, in Proceedings of the 4th International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 64–73.