AN EFFICIENT TWO-STAGE ITERATIVE METHOD FOR THE STEADY-STATE ANALYSIS OF MARKOV REGENERATIVE STOCHASTIC PETRI NET MODELS*

Luai M. Malhis and William H. Sanders
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{malhis, whs}@crhc.uiuc.edu

ABSTRACT

To enhance the modeling power of stochastic Petri nets (SPNs), new steady-state analysis methods have been proposed for nets that include non-exponential transitions. The underlying stochastic process is a Markov regenerative process (MRP) when at most one non-exponential transition is enabled in each marking. Time-efficient algorithms for constructing and solving the MRP have been developed. However, the space required to solve such models is often extremely large. This largeness is due to the large number of transitions in the MRP. Traditional analysis methods require that all these transitions be stored in primary memory for efficient computation. If the size of available memory is smaller than that needed to store these transitions, a time-efficient computation is impossible using these methods. To use this class of SPNs to model realistic systems, the space complexity of MRP analysis algorithms must be reduced. In this paper, we propose a new steady-state analysis method that is both time and space efficient. The new method takes advantage of the structure of the underlying process to reduce both computation time and required memory. The performance of the proposed method is compared to existing methods using several SPN examples.

Keywords: Markov Regenerative Stochastic Petri Nets, Deterministic Stochastic Petri Nets, Stochastic Activity Networks, Markov Chains, Iterative Solution Methods.

*This work was supported, in part, by NASA Grant NAG 1-1782.

I Introduction

The exponential assumption of stochastic Petri nets (SPNs) and extensions has been viewed as a major limitation in their modeling power for practical problems. Examples of non-exponential delays arise in modeling communication protocols where transmission time and time-outs are often deterministic. To enhance the modeling power of SPNs, new steady-state analysis methods have been proposed for nets with exponentially and generally distributed firing delays. For this class of Petri nets, steady-state analysis is possible if for each marking at most one transition with non-exponentially distributed delay is enabled. The first steady-state analysis algorithm for SPNs with exponentially distributed and deterministic firing delays was proposed by Ajmone-Marsan and Chiola [1]. Later, Lindemann [2] proposed a time-efficient algorithm for the steady-state analysis of SPNs with deterministic and exponential delays.

More recently, Choi, Kulkarni, and Trivedi [3] have defined a class of SPNs called Markov regenerative stochastic Petri nets (MRSPNs). A similar class of MRSPNs called extended deterministic stochastic Petri nets (EDSPNs) was also introduced in [4]. Both MRSPNs and EDSPNs allow at most one transition with non-exponentially distributed firing time to be enabled in each marking. Stochastic activity networks (SANs) [5] also allow the firing time on transitions (called activities) to be generally distributed. Algorithms for the analysis of MRSPN models are applicable to the analysis of SAN models that include non-exponentially distributed activities, and these algorithms have been implemented in [6]. The restriction on the enabling rules of non-exponential activities in SAN models follows from the restriction on non-exponentially distributed transitions in MRSPN models.

The underlying stochastic process of a MRSPN model is a Markov regenerative process (MRP) [7]. A MRP may not satisfy the Markov memoryless property in all states, but it has a sequence of embedded time points where this property is satisfied. At these time points, the future behavior of the process is independent of the past behavior. These time points are called regeneration points. In the steady-state analysis of a MRSPN model, the underlying MRP associated with a MRSPN model is first constructed, then analyzed. Time-efficient algorithms for constructing a MRP corresponding to a MRSPN models in which generally distributed transitions are deterministic, uniform, or "expolynomial" have been derived in [2, 4]. However, the number of transitions in the resulting MRP is often extremely large. For example, one simple model (illustrated in Section II) with only 3,888

states has 4,753,848 state transitions in its corresponding MRP. Traditional steady-state MRP analysis methods require that all these transitions be stored in memory for efficient computation. If the size of available memory is smaller than the size needed to store these transitions, memory becomes a bottleneck in the analysis phase. Therefore, for MRSPNs to be used in modeling realistic systems, the space complexity required of the MRP analysis algorithm must be reduced.

The goal of this investigation is to develop a space- and time-efficient MRP analysis algorithm. We do this by exploiting the structure of the MRP and utilizing disk storage in a smart way. In particular, we observe that the embedded Markov chain of a MRP typically has transition probabilities that differ by many orders of magnitude and, although not typically nearly completely decomposable (NCD), can be solved efficiently by decomposition of its state transition matrix into two parts. We propose a new "two-stage" algorithm to solve such models and show that it can dramatically reduce the amount of primary memory required for a solution, without significantly increasing computation time. Furthermore, we show that the method is significantly faster than standard methods (i.e., Gauss-Seidel) when sufficient memory is available. The new method permits the solution of much larger MRSPN models than was possible previously and makes practical the solution of measures defined on many realistic systems.

II Problem Definition

The construction of a MRP associated with a MRSPN model requires the computation of two matrices $\mathbf{P}$ and $\mathbf{C}$, as outlined in [2, 3, 4]. The matrix $\mathbf{P}$ represents an embedded Markov chain (EMC) and defines the transition probabilities between the states of the MRP. The $\mathbf{C}$ matrix represents the expected sojourn times in the states of the MRP between two regeneration points. To obtain the steady-state occupancy probability vector of the MRP, denoted π, we solve the following equations that involve the $\mathbf{P}$ and $\mathbf{C}$ matrices:

$$\nu(\mathbf{I} - \mathbf{P}) = 0, \quad \nu \cdot \mathbf{e} = 1, \text{ and} \tag{1}$$

$$\gamma = \nu \cdot \mathbf{C}, \pi = \frac{\gamma}{\gamma \cdot \mathbf{e}}, \tag{2}$$

where $\mathbf{I}$ denotes the identity matrix, $\mathbf{e}$ is a column vector of all ones, and ν, γ, and π are row vectors. Equation 1 is a system of linear equations, while Equation 2 is a matrix-vector multiplication and scaling. When Equation 1 is solved, vector ν contains the steady-state

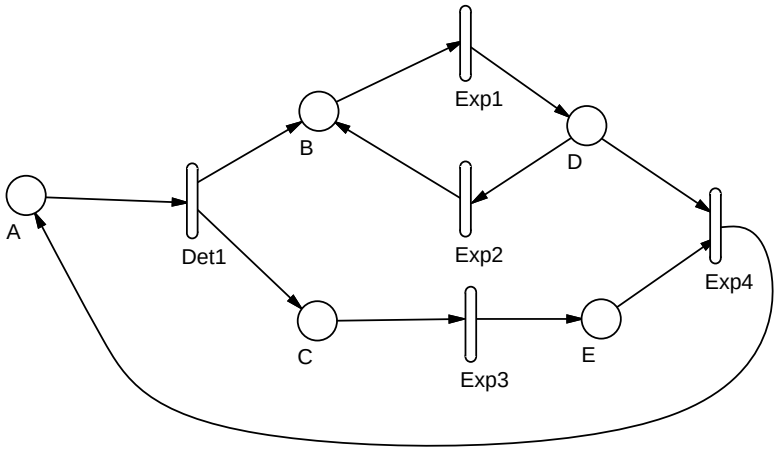

Figure 1: Molloy's Example with One Deterministic Activity

Table 1: Comparing Number of Non-zero Entries in the **Q**, **P**, and **C** Matrices for Different Tokens in A

Tokens initially in A	**States in CTMC**	**States in EMC**	**Non-zero entries in Q**	**Non-zero entries in P**	**Non-zero entries in C**	**Mbytes to store P and C**
12	819	698	4,303	146,693	158,691	3.67
14	1,240	1,071	6,630	351,736	377,525	8.75
16	1,785	1,560	9,673	754,988	804,865	18.72
18	2,470	2,181	13,525	1,486,998	1,576,126	36.76
20	3,311	2,950	18,291	2,734,355	2,884,091	67.42
22	4,324	3,883	24,058	4,753,848	4,993,231	116.97

occupancy probabilities of the EMC. By multiplying ν with **C** and then normalizing, we obtain π, the steady-state occupancy probabilities of the MRP.

To best illustrate the fill-in problem with MRSPN models, consider the small MRSPN example shown in Figure 1. This example was first considered by Molloy [8] to illustrate the applicability of SPN models. It shows fork, join, parallel, and sequential execution of a closed system. In Molloy's model, all transitions are exponentially distributed, while in our model, we make transition *Det1* deterministic.

Table 1 lists, for different initial markings of place A, the number of states in the continuous time Markov chain (CTMC) underlying the SPN model (if all activities are exponential), the reachable states in the EMC associated with the MRSPN, the number of non-zero entries in the matrix **Q** representing the CTMC, the number of non-zero entries

in the **P** and **C** matrices if transition *Det1* in Figure 1 is deterministic, and the memory required for the sparse storage of the **P** and **C** matrices. In computing required memory, we use sparse matrix representation such that for each row in the matrix we only keep track of the non-zero entries in that row and their corresponding column indices; the indices are of type long (4 bytes), and the non-zero entries are of type double (8 bytes). The amount of memory needed to store k non-zero entries is thus $12 \times k$ bytes. As the table shows, even though the number of reachable states is smaller in the EMC associated with the MRSPN than the SPN, the number of non-zero entries in **P** is much greater than the number of non-zero entries in **Q**. The degree of increase in the number of non-zero entries in a MRSPN model compared to the corresponding SPN model is dependent on the model itself and may not be so dramatic in other MRSPN models. However, in general, MRSPN models tend to generate much more dense state-transition matrices than corresponding SPN models.

For many MRSPN models, the storage requirement for **P** and/or **C** is often too large to fit in the main memory of a workstation. Determining the steady-state occupancy probabilities of the EMC (Equation 1) requires access only to the **P** matrix. In contrast, the **C** matrix is used in the computation of Equation 2, which is a single vector-matrix multiplication. Hence, the entries in the **C** matrix are accessed once and can be read from disk sequentially without too much overhead.

The memory bottleneck is in the solution of Equation 1. Traditional iterative solution methods (e.g, successive overrelaxation, Gauss-Seidel and the power methods) require access to all the entries in matrix **P** in each iteration. So with these methods, if the complete matrix cannot be stored in memory, storing some or all of the elements on disk increases computation time drastically. Therefore, an iterative method that efficiently utilizes disk and memory storage is needed to solve very large MRSPN models in a reasonable amount of time. In the next section, we will discuss the general properties of matrix **P** that arise when analyzing MRSPN models in order to motivate the development of a new iterative method.

III Properties of P

Observation of the **P** matrices generated from MRSPN models reveals that their entries typically differ by several orders of magnitude. The difference in the orders of magnitude between the entries in **P** is also a characteristic of a well-known class of Markov chains

known as nearly completely decomposable (NCD) chains. For Markov chains that are NCD, a class of iterative methods known as iterative aggregation and disaggregation (IAD) methods [9, 10, 11] can be used.

Thus, if the **P** matrix associated with a MRSPN model is NCD, then traditional IAD methods may be employed to obtain a space-efficient solution of the EMC, since IAD algorithms can be implemented such that all of the entries in the **P** matrix are not stored in memory simultaneously for efficient computation. To study the decomposability of MRSPN models, we followed the procedure suggested in [12]. This procedure sets all entries in **P** smaller than a decomposability factor δ to zero and searches for irreducible blocks. A block is irreducible if every state in the block can be reached from every other state within the block. Each irreducible block constitutes a block in a partition of the state space.

We first illustrate this procedure using a simple **M/D/1/6** queue. While simpler methods exist and should be used to solve this system, it serves to illustrate the problem with applying IAD methods to MRSPN models. Let the marking of the MRSPN model be the number of customers in the queue. The matrix **P** representing the EMC for this system is

$$\mathbf{P} = \begin{pmatrix} 0 & 1.0 & 0 & 0 & 0 & 0 & 0 \\ 0.368 & 0.368 & 0.184 & 0.0613 & 0.0153 & 0.00306 & 0.00034 \\ 0 & 0.368 & 0.368 & 0.184 & 0.0613 & 0.0153 & 0.0034 \\ 0 & 0 & 0.368 & 0.368 & 0.184 & 0.0613 & 0.0187 \\ 0 & 0 & 0 & 0.368 & 0.368 & 0.184 & 0.08 \\ 0 & 0 & 0 & 0 & 0.368 & 0.368 & 0.264 \\ 0 & 0 & 0 & 0 & 0 & 0.368 & 0.632 \end{pmatrix}.$$

Since the non-zero entries in each row i and column j are ordered such that $\mathbf{P}_{i,j} \geq \mathbf{P}_{i,j+1}$, the smallest entry in each row is the farthest from the diagonal. Thus, for this system, if the decomposability factor is smaller than or equal to 0.184, every state in the system is in the same block of the partition, and IAD methods cannot be used. On the other hand, if the decomposability factor is larger than 0.184, the system decomposes into six partitions. One partition contains two states, and each of the remaining partitions contains one state. As suggested by Stewart [12], IAD methods should not be used if the number of blocks is large and the number of states in each block is small. Thus, no benefit is gained in applying IAD methods in this case.

To study the decomposability of larger and more typical models, directed graph search algorithms were employed to find strongly connected components (irreducible blocks) in the EMC reachability graph. For example, this procedure was used to study the decomposability

of the MRSPN model given in Figure 1. As will be illustrated, none of the cases in Table 1 is decomposable into appropriately sized partitions. Specifically, consider the case when 16 tokens are placed in the place labeled A in Figure 1. The number of reachable states in the resulting MRSPN is $1,560$, and the number of non-zero entries in $\mathbf{P}$ is $754,988$. Setting all entries less than 1.0×10^{-4} ($474,649$ non-zero entries) to zero leaves only one strongly connected block in the partition. Furthermore, setting all entries less than 1.0×10^{-3} ($634,349$ non-zero entries) to zero results in partitioning the matrix into more than 243 blocks. Of the resulting partitions, 125 partitions contain a single state each, 104 partitions contain two states each, and only 14 partitions have more than two states assigned to them. The overhead involved in keeping track of the small partitions outweighs the advantages obtained in employing IAD methods for such models [12]. Selecting a decomposability factor less than 1.0×10^{-3} results in generating more one-state and two-state partitions. The decomposability of other MRSPN examples were studied and shown to be not decomposable.

Since the $\mathbf{P}$ matrix associated with many MRSPN models are not typically NCD, a new method that efficiently copes with the large number of non-zero entries in $\mathbf{P}$ is needed. We propose such a method in the next section.

IV Two-Stage Iterative Method

In this section, we describe a new two-stage iterative method to solve Equation 1. This method efficiently handles the large number of non-zero entries in $\mathbf{P}$.

Given an irreducible and stochastic matrix $\mathbf{P}$ representing an EMC associated with a MRSPN model, we need to solve the system of linear equations

$$\nu(\mathbf{I} - \mathbf{P}) = 0, \tag{3}$$

$$\nu\mathbf{e} = 1 \tag{4}$$

in order to obtain the steady-state state occupancy probabilities. Since the matrix $(\mathbf{I} - \mathbf{P})$ is singular, Equation 3 has many solutions. However, because $\mathbf{P}$ is irreducible and the solution must satisfy $\nu\mathbf{e} = 1$, the solution to (3) and (4) is unique. In the discussion that follows, we are interested in obtaining a solution to Equation 3. Let $\epsilon \in [0, 1]$ be a "decomposability factor." Based on the value of ϵ, we can decompose $\mathbf{P}$ into two matrices, $\mathbf{P}^l$ and $\mathbf{P}^s$, such that $\forall\ p_{ij} \in \mathbf{P},\ p_{ij} \in \mathbf{P}^s$ if $p_{ij} < \epsilon$, otherwise $p_{ij} \in \mathbf{P}^l$. Thus, $\mathbf{P} = \mathbf{P}^l + \mathbf{P}^s$, and Equation 3

can be rewritten as

$$\nu(\mathbf{I} - \mathbf{P}^l - \mathbf{P}^s) = 0, \text{ and, in turn,} \tag{5}$$

$$\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s. \tag{6}$$

Now suppose the row vector ν on the right-hand side of Equation 6 is known. Then the right-hand side of Equation 6 is another row vector $z = \nu\mathbf{P}^s$. Replacing the right-hand side of Equation 6 with z, we obtain the following system of linear equations:

$$\nu(\mathbf{I} - \mathbf{P}^l) = z. \tag{7}$$

Equation 7 has a unique solution because the matrix $(\mathbf{I} - \mathbf{P}^l)$ is nonsingular. $(\mathbf{I} - \mathbf{P}^l)$ is nonsingular since, by construction, $\mathbf{P}^l$ can be made to be a principal submatrix of an irreducible stochastic matrix. This is accomplished by adding a single row and a single column to $\mathbf{P}^l$. The entries in the extra column are chosen such that the modified matrix is stochastic. The entries in the extra row are chosen such that the modified matrix is irreducible. Thus, the modified matrix is irreducible and stochastic, with $\mathbf{P}^l$ as a principal submatrix. Then, according to a theorem in [13], since $\mathbf{P}^l$ is a principal submatrix of an irreducible stochastic matrix, $(\mathbf{I} - \mathbf{P}^l)^{-1}$ exists and is nonnegative, and hence $(\mathbf{I} - \mathbf{P}^l)$ is nonsingular.

Thus, if the exact ν were known, the solution of Equation 7 is the solution of Equation 6. In addition, since the matrix $\mathbf{P}^s$ is reduced to the vector z, we need only consider entries in $\mathbf{P}^l$ when Equation 7 is solved. However, since ν is not known *a priori*, ν must be given an initial approximation ν^0, which leads to the following two-stage iterative scheme:

$$z^k = \nu^k\mathbf{P}^s, \qquad k = 0, 1, 2, ... \tag{8}$$

$$\nu^k(\mathbf{I} - \mathbf{P}^l) = z^{k-1}, \qquad k = 1, 2, 3, ... \tag{9}$$

The iterative scheme has the following interpretation with respect to the matrix decomposition. If $\epsilon = 1$, then $\mathbf{P}^s = \mathbf{P}$ and $\mathbf{P}^l = 0$, and hence for $k > 0$, $\nu^k = z^{k-1} = \nu^{k-1}\mathbf{P}^s = \nu^{k-1}\mathbf{P}$. Therefore, $\nu^k = \nu^{k-1}\mathbf{P}$. Thus, when $\epsilon = 1$, the method reduces to the power method applied to solve Equation 3. If $\epsilon = 0$, then $\mathbf{P}^l = \mathbf{P}$ and $\mathbf{P}^s = 0$, and $z^k = 0, k \geq 0$, and hence the algorithm reduces to solving the system $\nu(\mathbf{I} - \mathbf{P}) = 0$ using whatever method is used to solve Equation 9. If $\mathbf{P}^s \neq 0$ and $\mathbf{P}^l \neq 0$, the case of interest to us, then the iteration process can be viewed as a power step and a solution step. For this case, the two

stages are executed as follows. For k = 0, let ν^0 be some initial solution vector and compute $z^0 = \nu^0 \mathbf{P}^s$. For $k > 0$, solve the system $\nu^k(\mathbf{I} - \mathbf{P}^l) = z^{k-1}$ and then compute z^k. This process continues until both vectors z and ν meet some stopping criterion. Note that in the iteration process, a different system of linear equations is solved in each iteration k.

Since Equation 9 is a linear system of equations, any of several iterative methods can be employed to solve Equation 9. Successive execution of Equations 8 and 9 requires the use of the entries in $(\mathbf{I} - \mathbf{P}^l)^T$ and $(\mathbf{P}^s)^T$ in each iteration k. As we discussed in Section II, for large MRSPN models, memory is a bottleneck, and we cannot store both matrices in memory at the same time. Recall that the decomposability factor partitions $\mathbf{P}$ into two disjoint matrices $\mathbf{P}^l$ and $\mathbf{P}^s$, such that $\mathbf{P} = \mathbf{P}^l + \mathbf{P}^s$ and $\mathbf{P}^s_{i,j} < \epsilon$, $\forall i, j = 1, ..., n$. The decomposability factor ϵ is normally chosen such that the entries in $\mathbf{P}^s$ are very small compared to the entries in $\mathbf{P}^l$. Thus, the entries in $\mathbf{P}^l$ will have a stronger relationship to the unknowns. Hence, if an iterative method is used to solve (9), we will iterate many times on (9) before applying (8). Thus, if memory storage is a bottleneck, we can store the entries in $\mathbf{P}^s$ on disk, while keeping the entries in $\mathbf{P}^l$ in memory. Since the number of outer iterations is typically very small compared to the number of inner iterations, disk access times are not too costly.

Numerical experiments [12, 14] have shown that the convergence rate of an iterative method to solve a system of the form $\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s$, where $(\mathbf{I} - \mathbf{P}^l)$ is nonsingular, is slow compared to solving an equivalent system based on a singular matrix. Thus, solving the system $\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s,\ \nu\mathbf{e} = 1$ iteratively may be slow. If we are able to transform this system to another system of the form $\tilde{\nu}(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0,\ \tilde{\nu}\mathbf{e} = 1$, where $(\mathbf{I} - \tilde{\mathbf{P}}^l)$ is singular, and if by solving the transformed system we can obtain the solution to the original system, the two-stage algorithm should be more efficient. We now provide such a transformation.

Given the decomposition $\mathbf{P}_{n\times n} = \mathbf{P}^l_{n\times n} + \mathbf{P}^s_{n\times n}$ and the system of linear equations,

$$\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s \tag{10}$$

$$\nu\mathbf{e} = 1, \tag{11}$$

we can construct the system

$$(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0, \tag{12}$$

$$(\tilde{\nu}, p_r)\mathbf{e} = 1, \tag{13}$$

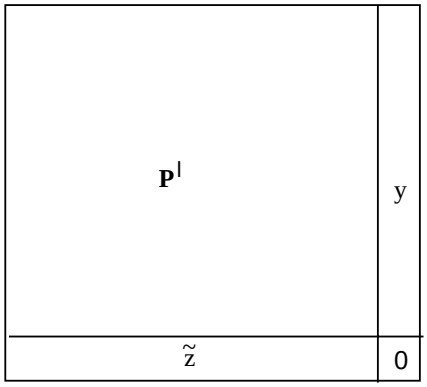

Figure 2: The $\tilde{\mathbf{P}}^l$ Matrix

where $\tilde{\mathbf{P}}^l$ is obtained by adding an extra state r into $\mathbf{P}^l$ as shown in Figure 2. The column vector $y = \mathbf{P}^s\mathbf{e}$ in Figure 2 is of size n, and it represents the transition probabilities from every state in $\mathbf{P}^l$ to the extra state r. Vector $\tilde{z}$ is defined as $\tilde{z} = \frac{\nu\mathbf{P}^s}{\nu\mathbf{P}^s\mathbf{e}}$. Vector $\tilde{z}$ (vector z normalized) is a row vector of size n, and it represents the transition probabilities from the extra state r into each state in $\mathbf{P}^l$. Vector $\tilde{z}$ has the constraint that $\tilde{z}\mathbf{e} = 1$, and it has a non-zero entry for each column in $\mathbf{P}^s$ that contains at least one non-zero entry.

Since vectors y and $\tilde{z}$ are chosen such that each row in $\tilde{\mathbf{P}}^l$ sums to one, it is stochastic. Furthermore, $\tilde{\mathbf{P}}^l$ is irreducible, since we assume $\mathbf{P}$ is irreducible and $\tilde{\mathbf{P}}^l{}_{i,r} > 0$ and $\tilde{\mathbf{P}}^l{}_{r,j} > 0$ for every pair of states i and j in $\mathbf{P}$ such that $\mathbf{P}^s_{i,j} > 0$. Therefore, every state in $\mathbf{P}$ that was reachable by a transition in $\mathbf{P}^s$ is still reachable in $\tilde{\mathbf{P}}^l$. Since $(\mathbf{I} - \tilde{\mathbf{P}}^l)$ is singular, the system $(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0$ has many solutions. However, since $\tilde{\mathbf{P}}^l$ is irreducible and the solution $(\tilde{\nu}, p_r)$ is subject to $(\tilde{\nu}, p_r)\mathbf{e} = 1$, the constrained solution is unique.

We next state and prove a theorem that shows that the steady-state probabilities of the system defined by Equations 3 and 4 can be obtained from the system defined by Equations 12 and 13.

Theorem 1 *Let* $\mathbf{P}$, $\tilde{\mathbf{P}}^l$, $\mathbf{P}^s$, ν, $\tilde{\nu}$, *and* $\tilde{z}$ *be as defined above. If* $(\tilde{\nu}, p_r)$ *is the solution to the modified system* $(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0$, $(\tilde{\nu}, p_r)\mathbf{e} = 1$, *then* $\nu = \frac{\tilde{\nu}}{\tilde{\nu}\mathbf{e}}$ *is the solution to the original system* $\nu(\mathbf{I} - \mathbf{P}) = 0$, $\nu\mathbf{e} = 1$.

Proof:
Suppose $(\tilde{\nu}, p_r)$ is the solution to $(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0$, $(\tilde{\nu}, p_r)\mathbf{e} = 1$, then (by simple manipu-

lation) $(\tilde{\nu}, p_r)\tilde{\mathbf{P}}^l = (\tilde{\nu}, p_r)$ holds. If we substitute $\tilde{\mathbf{P}}^l$ by its definition in the above equation, we obtain

$$(\tilde{\nu}, p_r) \begin{pmatrix} \mathbf{P}^l & y \\ \tilde{z} & 0 \end{pmatrix} = (\tilde{\nu}, p_r),$$

which can be written equivalently as $\tilde{\nu}\mathbf{P}^l + p_r\tilde{z} = \tilde{\nu}$ and $\tilde{\nu}y + 0 = p_r$. From the second equation, we can substitute $p_r = \tilde{\nu}y$ in the first equation to obtain $\tilde{\nu}\mathbf{P}^l + \tilde{\nu}y\tilde{z} = \tilde{\nu}$, and hence $\tilde{\nu}(\mathbf{I} - \mathbf{P}^l) = \tilde{\nu}y\tilde{z}$. If we replace the vector y in this equation by its definition, $y = \mathbf{P}^s\mathbf{e}$, we obtain

$$\tilde{\nu}(\mathbf{I} - \mathbf{P}^l) = \tilde{\nu}\mathbf{P}^s\mathbf{e}\tilde{z}.$$

Then, we substitute $\tilde{\nu}$ by its definition from the theorem, $\tilde{\nu} = \nu\tilde{\nu}\mathbf{e}$, to obtain $\nu\tilde{\nu}\mathbf{e}(\mathbf{I} - \mathbf{P}^l) = \nu\tilde{\nu}\mathbf{e}\mathbf{P}^s\mathbf{e}\tilde{z}$. The result of the row vector $\tilde{\nu}$ and the column vector $\mathbf{e}$ multiplication is a scalar. Thus, the equation can be simplified by removing the scalar $\tilde{\nu}\mathbf{e}$ from both sides,

$$\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s\mathbf{e}\tilde{z}.$$

Then, by definition, $\tilde{z} = \frac{\nu\mathbf{P}^s}{\nu\mathbf{P}^s\mathbf{e}}$. If we replace $\tilde{z}$ by its definition in the above equation, we obtain

$$\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s\mathbf{e}\frac{\nu\mathbf{P}^s}{\nu\mathbf{P}^s\mathbf{e}},$$

which is equivalent to $\nu(\mathbf{I} - \mathbf{P}^l) = \nu\mathbf{P}^s$, and likewise $\nu(\mathbf{I} - \mathbf{P}^l - \mathbf{P}^s) = 0$. Since by definition the entries of $\nu = \frac{\tilde{\nu}}{\tilde{\nu}\mathbf{e}}$ sum to one, and $\mathbf{P} = \mathbf{P}^l + \mathbf{P}^s$, ν is the solution to the original system of equations

$$\nu(\mathbf{I} - \mathbf{P}) = 0,\ \nu\mathbf{e} = 1\ \square.$$

The splitting of a matrix $\mathbf{P}$ into two matrices $\mathbf{P}^l$ and $\mathbf{P}^s$ and adding a new state to $\mathbf{P}^l$ to obtain $\tilde{\mathbf{P}}^l$ was first proposed by Franceschinis and Muntz [15, 16] to transform a quasi-lumpable Markov chain into a lumpable chain. Their method is based on the observation that a system may contain objects that exhibit symmetric behavior from a qualitative point of view, but that symmetries disappear when quantitative aspects are considered. An example of such a system is a multi-server system with very small differences in the service rate. In their method, a CTMC represented by a matrix $\mathbf{Q}_{n\times n}$ is modified such that $\mathbf{Q} = \mathbf{Q}^- + \mathbf{Q}^\epsilon$, where $\mathbf{Q}^-$ meets the strong lumpability condition and $\mathbf{Q}^\epsilon_{ij} < \epsilon\ \forall i, j = 1, 2, .., n$. In their method, *a priori* knowledge about the symmetries in the model is required to obtain

$\mathbf{Q}^-$. Then the entries in $\mathbf{Q}$ are manipulated by adding or subtracting values with magnitude less than ϵ to get $\mathbf{Q}^-$ and $\mathbf{Q}^\epsilon$ such that $\mathbf{Q} = \mathbf{Q}^- + \mathbf{Q}^\epsilon$ and $\mathbf{Q}^-$ is strongly lumpable. Then an extra state is added to $\mathbf{Q}^-$ such that the transitions into the new state from every state in $\mathbf{Q}^-$ is the row sum of $\mathbf{Q}^\epsilon$. The modified system is then aggregated into a smaller system, and performance bounds are computed from the aggregated system.

We use their idea of splitting the matrix into two matrices and adding the extra state as discussed above. However, in our method, no manipulation of the entries in the matrix is required, and we compute exact performance measures instead of performance bounds. The method proposed by Franceschinis and Muntz is intended to reduce the size of the underlying state space; the two-stage method is intended to efficiently handle the large number of transitions between the states.

In the next section, we describe the two-stage iterative algorithm that solves the linear system defined by Equations 12 and 13 in the inside iteration and computes ν and $\tilde{z}$ in the outside iteration.

V Algorithm Description

Theorem 1 states that computing vectors $\tilde{z}$ and y as defined and solving the modified matrix $\tilde{\mathbf{P}^l}$ shown in Figure 2 gives the solution to the original system defined by the matrix $\mathbf{P} = \mathbf{P}^l + \mathbf{P}^s$. However, when the matrix $\tilde{\mathbf{P}^l}$ is first set up, the vector $\tilde{z}$ is unknown. This vector contains the transition probabilities from the extra state r to the states in the EMC. Any column in $\mathbf{P}^s$ with all zero elements corresponds to a zero element in $\tilde{z}$. The values of the remaining elements of $\tilde{z}$ are given an initial approximation $\tilde{z}^0 = \frac{\nu^0 \mathbf{P}^s}{\nu^0 \mathbf{P}^s \mathbf{e}}$, where $\nu_i^0 = 1/n \ \forall i = 1, 2, ...n$ is an initial approximation to the final solution vector ν. This leads to a two-stage iterative scheme to compute $\tilde{z}$ and to solve the linear system of equations $(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}^l}) = 0$, subject to $(\tilde{\nu}, p_r)\mathbf{e} = 1$. The proposed two-stage iterative algorithm follows.

Algorithm 1 (Two-state iterative algorithm)

1. *Initialization step:*

   Partition $\mathbf{P}$ *into* $\mathbf{P}^l$ *and* $\mathbf{P}^s$ *and set*

   $y = \mathbf{P}^s \mathbf{e}$,

   $\tilde{\mathbf{P}^l}_{i,j} = \mathbf{P}^l_{i,j}, \ \forall i = 1, ..., n, \ \ j = 1, ..., n,$

$\tilde{\mathbf{P}}^l{}_{i,n+1} = y_i, \ \forall i = 1, ..., n,$

$\nu_i^0 = 1/n \ \forall i = 1, ...n,$

$\tilde{z}^0 = \frac{\nu^0 \mathbf{P}^s}{\nu^0 \mathbf{P}^s \mathbf{e}}$, *and*

$k = 0, m = 0$

2. *Modify the* $\tilde{\mathbf{P}}^l$ *matrix using the latest values for* $\tilde{z}$.

   $\tilde{\mathbf{P}}^l{}_{n+1,j} = \tilde{z}_j^k \ \forall j = 1, ..., n$

3. *Solve the system* $(\tilde{\nu}, p_r)(\mathbf{I} - \tilde{\mathbf{P}}^l) = 0$ *as follows:*

   (a) *Construct the iteration matrix* $\mathbf{H}$ *by splitting* $(\mathbf{I} - \tilde{\mathbf{P}}^l)^T$ *for some iterative method.*

   (b) *Do an iteration by computing*

   $((\tilde{\nu}, p_r)^{m+1})^T = \mathbf{H}((\tilde{\nu}, p_r)^m)^T.$

   (c) *Conduct a local test of convergence on vector* $(\tilde{\nu}, p_r)^{m+1}$. *If it meets the chosen stopping criterion, go to step 4; otherwise set* $m = m + 1$ *and go to b.*

4. *Compute new values of* ν *and* $\tilde{z}$.

   $k = k + 1$, *and*

   $\nu^k = \frac{\tilde{\nu}^{m+1}}{\tilde{\nu}^{m+1}\mathbf{e}}$, $\tilde{z}^k = \frac{\nu^k \mathbf{P}^s}{\nu^k \mathbf{P}^s \mathbf{e}}$.

5. *Conduct a global test of convergence on vectors* ν^k *and* $\tilde{z}^k$. *If both vectors meet the chosen stopping criterion, quit and take* ν^k *to be the solution vector; otherwise go to Step 2.*

Note that the algorithm does not specify the particular iterative method to be used to solve the system of linear equations in Step 3. Because large matrices are usually encountered, iterative methods such as successive overrelaxation (SOR) or Gauss-Seidel are the natural choices. In our implementation, Gauss-Seidel was selected for the inside iteration process. In Step 3, the selected iterative method repeats until the stopping criterion on the vector $(\tilde{\nu}, p_r)$ is met. In our implementation, the Cauchy criterion was selected as the iteration stopping criterion. When the iteration process in Step 3 stops, new values for ν are determined and then new values for $\tilde{z}$ are computed as shown in Step 4. The matrix $\tilde{\mathbf{P}}^l$ is updated to reflect the changes in vector $\tilde{z}$ as shown in Step 2. Then the iteration process

in Step 3 continues with the latest $(\tilde{\nu}, p_r)$ vector as the initial vector. The two stages of the algorithm can be viewed as a solution stage, Step 3, and a correction stage, Steps 2 and 4.

As with the case of many iterative methods, a proof that the algorithm converges is not known for all iterative methods used in Step 3. Specific cases where convergence can be proved are discussed in [17]. In spite of this lack of proof, we have not encountered a decomposability factor $\epsilon \in [0, 1]$ or a MRSPN model where the algorithm does not converge.

VI Inside and Outside Iteration Execution

The chosen matrix decomposition affects the computation time of the algorithm in a complex manner, which depends on the nature of the $\mathbf{P}$ matrix. In particular, the larger ϵ is, the greater the number of elements stored in $\mathbf{P}^s$. Larger values of ϵ thus increase the computation cost of an outside iteration and the number of outside iterations needed for convergence. On the other hand, the fewer the number of elements in $\tilde{\mathbf{P}}^l$, the smaller the computation cost per inside iteration.

The difference in magnitude between the elements in $\mathbf{P}^s$ and $\mathbf{P}^l$ is also important. Specifically, since the entries in $\mathbf{P}^l$ are much larger in magnitude than the entries in $\mathbf{P}^s$, they have a stronger relationship to the unknowns. The number of inside iterations should thus exceed the number of outside iterations. Since the relative importance of these factors is unknown in general, and depends on the entries of the specific $\mathbf{P}$, finding the optimal number of inside iterations to execute for each outside iteration to minimize the total execution time is difficult (if not impossible).

We have experimented with many policies. An example of a simple policy we experimented with is to do a fixed (small to large) number of inner iterations per outer iteration. However, since different models have different convergence rates, the performance of the algorithm with this policy will be dependent on the model itself and the decomposability factor. The most efficient and consistent policy we found is to take into consideration the relative computation cost of a single inside iteration to the combined computation cost of a single inside iteration and a single outside iteration. In this policy, if we let the error at the beginning of a sequence k of inside iterations be α_k^b, then we require the error at the end of the sequence be $\alpha_k^e = \alpha_k^b \times \beta$, where $\epsilon \leq \beta \leq 1.0$. The parameter β is always larger than or equal to ϵ to guard against β being too small and, hence, causing the execution of too many inside iterations per outside iteration.

If we let $elem(\mathbf{A})$ denote the number of non-zero elements in matrix $\mathbf{A}$, we choose β such that

$$\beta = Max(\frac{elem(\tilde{\mathbf{P}}^l)}{elem(\tilde{\mathbf{P}}^l) + elem(\mathbf{P}^s)}, \epsilon). \tag{14}$$

Thus, the parameter β is determined based on the decomposability factor ϵ and the cost ratio of executing an inside iteration to the combined cost of executing a single inside iteration and a single outside iteration.

In our implementation, we use the Cauchy criterion as a measure of the "error" in the solution vector. Let the Cauchy criterion, defined as

$$\alpha_k^m = Max_i \mid (\tilde{\nu}, p_r)_i^m - (\tilde{\nu}, p_r)_i^{m-1} \mid \quad \forall i = 1, 2, .., n+1, \tag{15}$$

represent the error in the solution vector $(\tilde{\nu}, p_r)$ after executing m inside iterations in the kth inside iteration. Thus, if the error at the beginning of sequence k is α_k^b and the inside iteration is executed until $\alpha_k^e \leq \alpha_k^b \times \beta$, then the error is reduced by a factor of β.

In this approach, for each sequence, $k = 1, 2, 3...$, of inside iterations, we must compute α_k^b and α_k^e. For the initial sequence, $k = 1$, $\alpha_1^b = 1$, and $\alpha_1^e = \epsilon$. For each subsequent sequence, $k > 1$, execute the inside iteration twice and set $\alpha_k^b = \alpha_k^2$, then execute the inside iteration m additional iterations until $\alpha_k^m \leq \alpha_k^2 \times \beta$. In other words, the inside iteration is first executed twice, and the Cauchy criterion is computed to estimate the error at the beginning of this sequence α_k^b. Then, the inside iteration is executed a number of times until the Cauchy criterion is smaller than or equal to $\alpha_k^b \times \beta$.

This policy has shown consistent and efficient execution behavior across several examples and decomposability factors. In the next section, the performance of the implementation of the algorithm is discussed.

VII Algorithm Evaluation

In this section, the computation time and memory usage of the two-stage method will be demonstrated using two examples. A third example can be found in [17]. We use SANs as our SPN representation, since we are familiar with them and have implemented the method as a solver in *UltraSAN* [18]. All the runs were done on a Hewlett Packard workstation model 715/64 with 160 Megabytes of RAM. For all examples, the Gauss-Seidel method was used as the inside iterative method.

Table 2: Decomposition of $\mathbf{P}$ and Memory Requirements as a Function of ϵ, Molloy's Example with 18 Tokens

Decomp. factor (ϵ)	**Elements in $\tilde{\mathbf{P}}^l$**	**Elements in $\mathbf{P}^s$**	**Bytes alloc. for $\tilde{\mathbf{P}}^l$**	**% Elements of P in $\mathbf{P}^s$**
0.0	1,486,998	0	18.0M	0.0
1.0×10^{-6}	1,011,999	478,838	12.2M	32.2
1.0×10^{-5}	792,783	698,130	9.5M	47.0
1.0×10^{-4}	501,055	989,871	6.1M	66.6
1.0×10^{-3}	206,722	1,284,205	2.5M	86.4
1.0×10^{-2}	40,884	1,450,043	0.5M	97.5
1.0×10^{-1}	7,767	1,483,160	0.1M	99.7

To evaluate the performance of the algorithm, we compared solution of the generated system of linear equations using the Gauss-Seidel and the power methods to solution of the decomposed and modified system using the two-stage method. The iteration process was stopped when the Cauchy criterion on the final solution vector was smaller than 10^{-9}. Thus, execution of the Gauss-Seidel and power methods was stopped when the maximum difference between two successive iterates on the solution vector ν was smaller than 10^{-9}. Execution of the two-stage method was stopped when the maximum difference between two successive iterates on both vectors ν and $\tilde{z}$ was smaller than 10^{-9}. In the following discussion, the original chain refers to the EMC represented by the matrix $\mathbf{P}$, and the modified chain refers to the EMC represented by the matrices $\tilde{\mathbf{P}}^l$ and $\mathbf{P}^s$.

The first example presented is the one we used to demonstrate the fill-in problem with MRSPN models, shown in Figure 1. Consider the case when 18 tokens are initially in A. For this model, the number of non-zero entries in $\mathbf{P}$ is 1,486,998, and the number of bytes needed to store $\mathbf{P}$ (using the sparse matrix representation method described earlier) is 18 Megabytes. Table 2 lists different decomposability factors and the corresponding number of non-zero entries in $\tilde{\mathbf{P}}^l$ and $\mathbf{P}^s$ for that decomposability factor. In addition, the amount of memory needed to store $\tilde{\mathbf{P}}^l$ for each decomposability factor is also listed. For example, if $\epsilon = 1.0 \times 10^{-6}$, the number of non-zero entries in $\tilde{\mathbf{P}}^l$ is 1,011,999, and 12.2 Megabytes of memory are needed to store $\tilde{\mathbf{P}}^l$. The number of non-zero entries in $\mathbf{P}^s = 478,838$ (32.2% of the total number of entries in $\mathbf{P}$).

Figure 3 shows the elapsed computation time, as a function of ϵ, for the Gauss-Seidel,

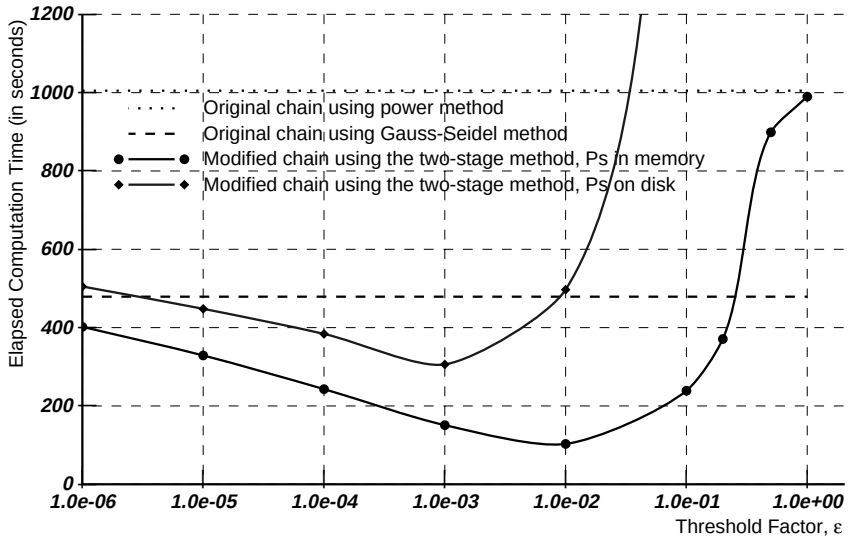

Figure 3: Elapsed Computation Time as a Function of ϵ, with $\mathbf{P}^s$ on Disk and in Memory

power, and two-stage methods. If the Gauss-Seidel and power methods are used to solve the original chain, the complete $\mathbf{P}$ matrix must be stored in memory. If the two-stage method is used to solve the modified chain, two cases are considered. The first case is storage of both $\tilde{\mathbf{P}}^l$ and $\mathbf{P}^s$ in memory. The second case is storage of $\tilde{\mathbf{P}}^l$ in memory and $\mathbf{P}^s$ on disk. As shown in Figure 3, when the Gauss-Seidel and power methods are used to solve the original chain, the elapsed computation times are 479 and 1005 seconds, respectively (no decomposition is done, so there is no dependence on ϵ). When the two-stage method is used to solve the modified chain, the elapsed computation time is dependent on ϵ and the storage location of $\mathbf{P}^s$.

More specifically, when $\mathbf{P}^s$ is in memory and $\epsilon \leq 1.0 \times 10^{-2}$, Figure 3 shows that the two-stage method's elapsed computation time decreases as larger values of ϵ are selected. This follows from Table 2, in which the number of non-zero entries in $\tilde{\mathbf{P}}^l$ decreases as ϵ increases. Hence, the computation cost of an inside iteration decreases. To better understand why the two-stage method is faster than the Gauss-Seidel and power methods for these values of ϵ, we compare the cost of an iteration and the number of iterations executed by the Gauss-Seidel, power, and two-stage methods. Figure 4 shows, for each ϵ, the total number of inner iterations executed by the two-stage method, the total number of outer iterations executed by the two-stage method, the total number of iterations executed by the Gauss-Seidel

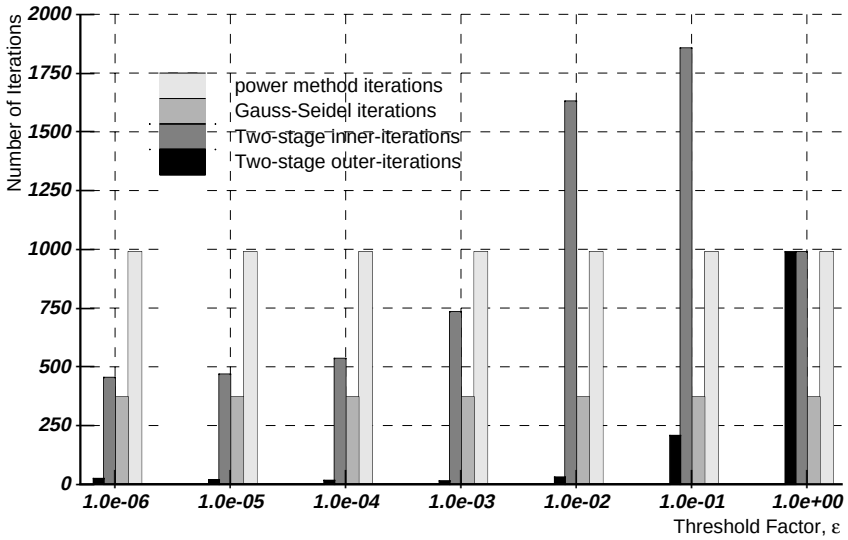

Figure 4: Number of Iterations as a Function of ϵ

method, and the total number of iterations executed by the power method. Figure 4 shows that the Gauss-Seidel and power methods executed 372 and 990 iterations, respectively. In the two-stage method, the number of inner iterations and outer iterations executed is dependent on ϵ.

For example, selecting $\epsilon = 1.0 \times 10^{-2}$, Table 2 shows that 2.5% of the elements in $\mathbf{P}$ are involved in the inside iteration, while 97.5% of the elements in $\mathbf{P}$ are involved in the outside iteration. Figure 4 shows that for $\epsilon = 1.0 \times 10^{-2}$ the two-stage method executed a total of 1630 inside iterations and 31 outside iterations compared to 372 for the Gauss-Seidel method and 990 for the power method. In all methods, the cost of an iteration is linearly proportional to matrix size. Therefore, the cost of a Gauss-Seidel iteration equals the cost of an iteration in the power method, while at $\epsilon = 1.0 \times 10^{-2}$, the cost of each inside iteration in the two-stage method is 2.5% the cost of a single Gauss-Seidel iteration, and the cost of each outside iteration is 97.5% the cost of a single Gauss-Seidel iteration. Thus, when $\epsilon = 1.0 \times 10^{-2}$, the total cost of all of the two-stage method inside iterations is $1630 \times 0.025 = 42$ Gauss-Seidel iterations. The total cost of all of the two-stage method outside iterations is 31 Gauss-Seidel iterations. Therefore, the total cost of executing the two-stage method is 73 Gauss-Seidel iterations. This makes the two-stage method five times faster than Gauss-Seidel at $\epsilon = 1.0 \times 10^{-2}$ as shown in Figure 3.

The elapsed computation time of the two-stage method does not continue to decrease

as larger values of ϵ are selected because for relatively large values of ϵ both the cost of an outside iteration and the number of outside iterations executed increase. As shown in Figure 3, for $\epsilon > 1.0 \times 10^{-2}$, the two-stage method's elapsed computation time is worse than the best case ($\epsilon = 1.0 \times 10^{-2}$). For values of $\epsilon > 1.0 \times 10^{-2}$, the number of outside iterations executed is much larger than the number of outside iterations executed when $\epsilon = 1.0 \times 10^{-2}$. For example, if $\epsilon = 1.0 \times 10^{-1}$, the two-stage method executed 207 outer iterations, and each outer iteration involved more than 98% of the elements in $\mathbf{P}$. However, at this value of ϵ, the two-stage method still outperforms Gauss-Seidel because the total number of outside iterations is still less than the total number of Gauss-Seidel iterations. This trend continues until $\epsilon = 1.0$, at which point the two-stage method behaves like the power method. At this ϵ (see Figure 4), a single inside iteration is executed for each outside iteration, and the total number of outside iterations executed equals the total number of iterations executed by the power method.

When $\mathbf{P}^s$ is stored in memory and $\epsilon = 1.0 \times 10^{-2}$, the two-stage method is five times faster than Gauss-Seidel, but the memory required in both methods is the same. If $\mathbf{P}^s$ is stored on disk, memory usage in the two-stage method is reduced at the expense of longer elapsed computation time compared to having $\mathbf{P}^s$ in memory. However, solving the modified chain using the two-stage method with $\mathbf{P}^s$ on disk and the number of outside iterations executed is smaller than 30, i.e., $1.0 \times 10^{-5} \leq \epsilon < 1.0 \times 10^{-2}$, is faster than solving the original chain using the Gauss-Seidel method.

For example, if $\epsilon = 1.0 \times 10^{-3}$, the modified chain is solved in 306 seconds and required 2.5 Megabytes of memory to store $\tilde{\mathbf{P}}^l$. In this case, the number of inside iterations totaled 734, and the number of outside iterations totaled 14. When solving the original chain using Gauss-Seidel, 18 Megabytes of memory storage are needed to hold $\mathbf{P}$, and 479 seconds of computation time is needed to obtain convergence. At $\epsilon = 1.0 \times 10^{-3}$, the two-stage method requires less than 14% of memory storage and achieves 33% reduction in computation time compared to Gauss-Seidel. For values of ϵ greater than 1.0×10^{-3}, small additional savings in memory are achieved, but computation times are longer. For values of $\epsilon > 1.0 \times 10^{-3}$, more outside iterations are executed and a larger percentage of $\mathbf{P}$ is accessed in each outside iteration.

For this example, the two-stage method thus requires significantly less memory than Gauss-Seidel when $\mathbf{P}^s$ is on disk and is faster than Gauss-Seidel when $\epsilon < 1.0 \times 10^{-2}$. To

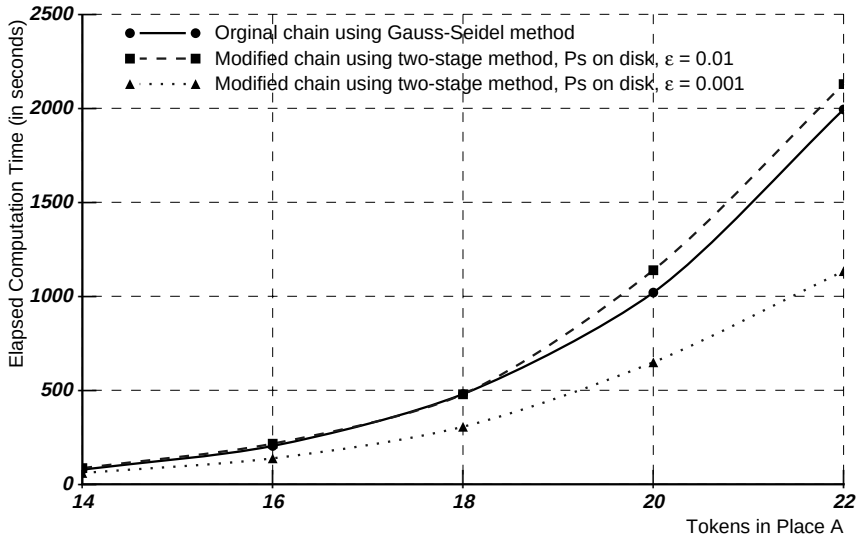

Figure 5: Elapsed Computation Time as a Function of Tokens, $\mathbf{P}^s$ on Disk

further demonstrate the memory savings that can be achieved using the two-stage method, we scaled the model shown in Figure 1 by placing a different number of tokens in place A (see Table 1). As Table 1 indicates, as larger models are generated, the size of $\mathbf{P}$ sharply increases, and memory storage quickly becomes a bottleneck in the solution process. For example, if the number of tokens in place A is 22, then 57 Megabytes are needed to store $\mathbf{P}$.

Figures 5 and 6, respectively, compare the elapsed computation time and the memory usage of the two-stage method when $\mathbf{P}^s$ is on disk to Gauss-Seidel. The solid curve in Figures 5 and 6 shows the elapsed computation time and memory usage of the Gauss-Seidel method. The other two curves in each figure show elapsed computation time and memory usage of the two-stage method when $\epsilon = 1.0 \times 10^{-3}$ and $\epsilon = 1.0 \times 10^{-2}$ and $\mathbf{P}^s$ is stored on disk. For example, when 22 tokens are placed in A and $\epsilon = 1.0 \times 10^{-3}$, the modified chain is solved in 1134 seconds, and $\tilde{\mathbf{P}}^l$ occupies 5.8 Megabytes. If $\epsilon = 1.0 \times 10^{-2}$, the modified chain is solved in 2129 seconds with 0.9 Megabytes allocated for $\tilde{\mathbf{P}}^l$. It took 1994 seconds to solve the original chain using Gauss-Seidel, and 57 Megabytes of memory storage was required to hold $\mathbf{P}$. Solving this model using the two-stage iterative method with $\mathbf{P}^s$ on disk and selecting $\epsilon = 1.0 \times 10^{-3}$ requires only 10% of the memory required by Gauss-Seidel and is 44% faster than Gauss-Seidel. Thus, when memory storage is limited, the

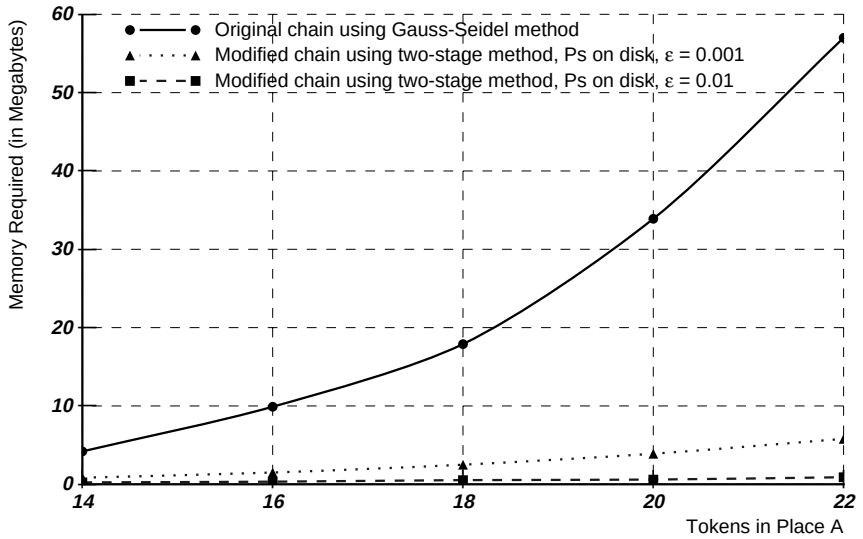

Figure 6: Memory Usage as Function of Tokens, $\mathbf{P}^s$ on Disk

two-stage method permits the solution of very dense MRSPN models that are much larger than available memory, with no increase in elapsed computation time.

Another example is a SAN model of a polling system [19] shown in Figure 7. This is a finite buffer, single server, exhaustive service polling system model. In this model, the server polls a node, and if there are customers waiting to be served at that node, the server starts serving the waiting customers. If there are no customers waiting, the server starts polling the next node. Nodes are polled in circular fashion. When a server starts serving customers at a given node, the server polls the next node only when no more customers are left to be served at that node. A place labeled P_i in Figure 7 contains the number of empty buffers at node i. A token in place labeled PS_i signifies that the server is polling node i. A token in place labeled S_i signifies that the server is serving customers at node i. An input gate labeled IG_i contains the function that enables the corresponding instantaneous activity I_i. The enabling function for gate IG_i is that there are no more customers waiting to be served in Que_i and the server is at node i. In this model, we assumed the polling time of a node is deterministic, and the service time is also deterministic. Customer inter-arrival time at a node is assumed to be exponential. Finally, all nodes have an equal number of empty buffers, and we varied this number to get different models.

Table 3 lists the initial number of empty buffers at each node, the number of reachable markings, the number of reachable states in P, and the number of non-zero entries in matrix

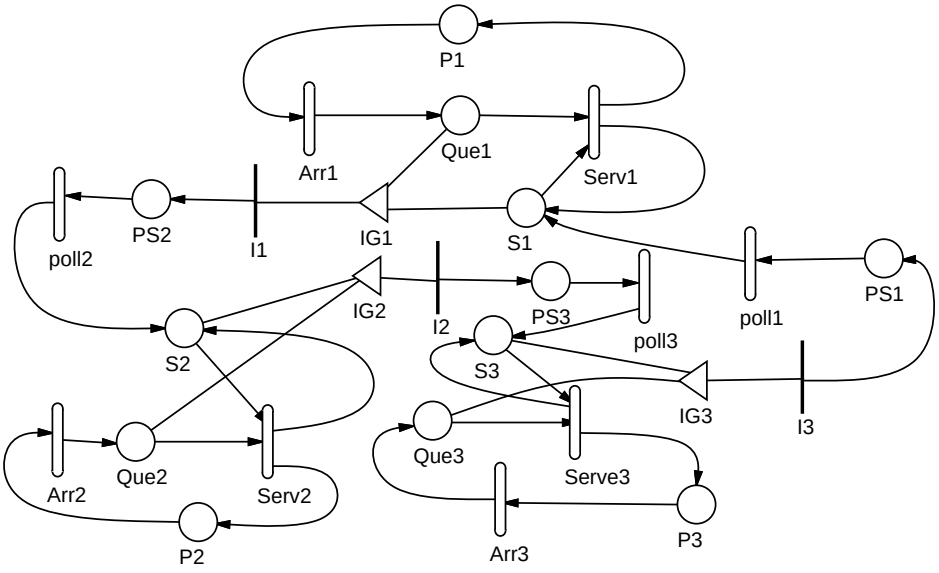

Figure 7: Polling System Model

Table 3: Polling Model, State Space and **P** Matrix Sizes

Maximum available buffers	**Number of states in CTMC**	**Number of reachable states**	**Non-zero entries in P**
11	9,936	5,184	1,274,397
12	12,675	6,591	1,956,048
13	15,876	8,232	2,879,625
14	19,575	10,125	4,086,222
15	23,808	12,288	5,614,161

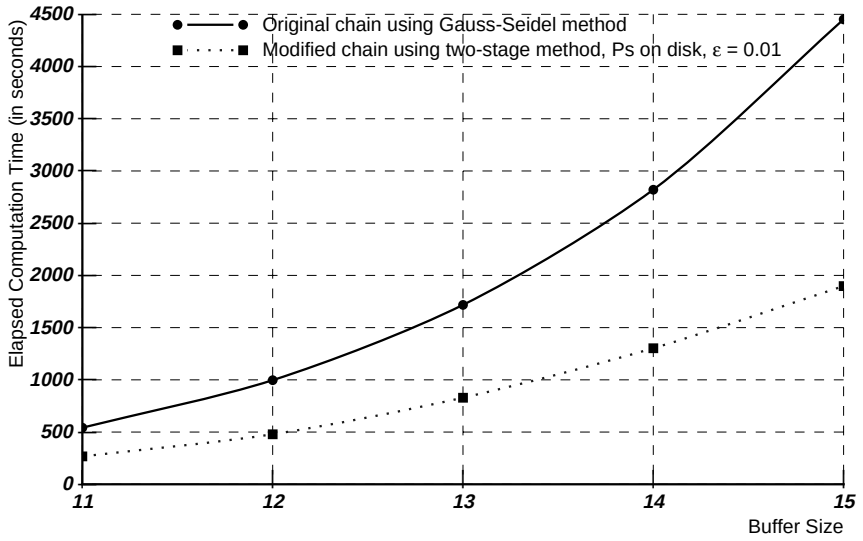

Figure 8: Elapsed Computation Time as a Function of Buffer Size, $\mathbf{P}^s$ on Disk

$\mathbf{P}$. As the table indicates, the number of non-zero entries in $\mathbf{P}$ grows very quickly as larger models are analyzed.

The efficiency of the two-stage method for solving very large models is shown in Figures 8 and 9. For example, when the buffer size at each node is set to 22, the $\mathbf{P}$ matrix contains more than 5.5×10^6 non-zero entries. In this model, as Figures 8 and 9 indicate, when selecting $\epsilon = 1.0 \times 10^{-2}$, the two-stage method is twice as fast as Gauss-Seidel and requires less than 10% of the memory storage required by Gauss-Seidel. Once again, the two-stage method is shown to be very efficient in solving models that are too large to fit in memory.

VIII Summary and Conclusion

In this paper, we have demonstrated the fill-in problem that accompanies large MRSPN models. We discussed the properties of the $\mathbf{P}$ matrix associated with these models that make them suitable to be solved using a two-stage iterative scheme. We then proposed a new two-stage iterative method that efficiently handles an EMC associated with a MRSPN model by utilizing disk storage when needed. We devised a time-efficient algorithm, implementing the method, that executes more inner iterations than outer iterations. In addition, we provided an implementation of the two-stage iterative method and demonstrated both the time and space efficiency of this algorithm using several MRSPN examples.

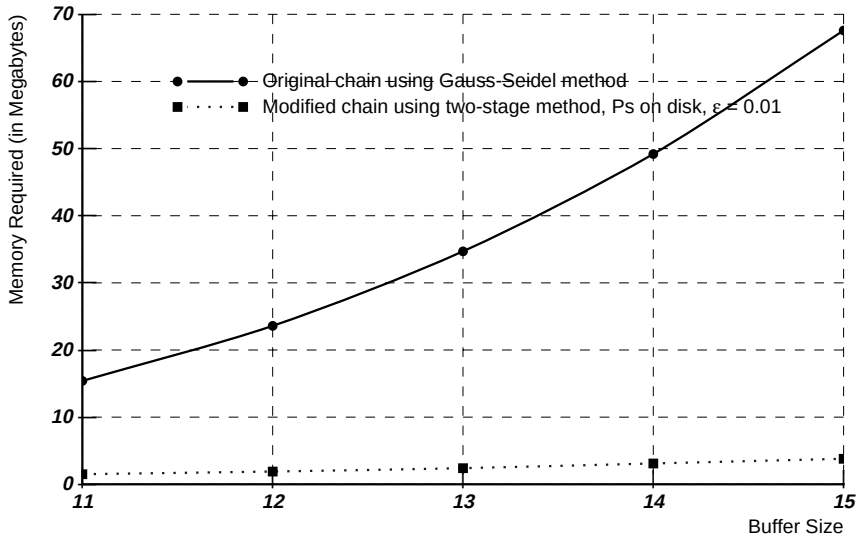

Figure 9: Memory Usage as a Function of Buffer Size, $\mathbf{P}^s$ on Disk

The two-stage method is time efficient because it iterates more on the large-valued entries than on the small-valued entries. It is space efficient because it utilizes disk to store the small-valued entries if memory is limited. Experimental results showed that if memory storage is not a problem, the two-stage method could be five times faster than traditional methods. Furthermore, if memory is limited, utilizing disk reduces required memory storage greatly at a small increase in elapsed computation time compared to unlimited memory. Therefore, the two-stage method is a more appropriate method for the steady-state analysis of MRSPN models. It is faster and more memory efficient than other known methods.

REFERENCES

[1] M. Ajmone-Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," in *Lecture Notes in Computer Science*, vol. 266, pp. 132–145, Springer-Verlag, 1987.

[2] C. Lindemann, "An improved numerical algorithm for calculating steady-state solution of deterministic and stochastic Petri net models," *Performance Evaluation*, vol. 18, 1993.

[3] H. Choi, V. Kulkarni, and K. S. Trivedi, "Markov regenerative stochastic Petri nets," *Performance Evaluation*, vol. 20, pp. 337–357, 1994.

[4] G. Ciardo, R. German, and C. Lindemann, "A characterization of the stochastic process underlying a stochastic Petri net," *IEEE Transactions on Software Engineering* , vol. 20, no. 7, pp. 506-515, July, 1994.

[5] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proceedings of the International Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.

[6] B. P. Shah, *Analytic solution of stochastic activity networks with exponential and deterministic activities*. Master's Thesis, University of Arizona, Tucson, Arizona, August 1993.

[7] V. Kulkarni, *Modeling and Analysis of Stochastic Systems*. Chapman-Hall, 1995.

[8] M. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. 24, pp. 913–917, September 1982.

[9] M. Haviv, "An aggregation/disaggregation algorithm for computing the stationary distribution of a large Markov chain," *Communications in Statistics—Stochastic Models*, vol. 8, pp. 565–575, 1992.

[10] R. Koury, D. F. McAllister, and W. J. Stewart, "Methods for computing stationary distributions of nearly-completely-decomposable Markov chains," *SIAM Journal of Algebraic and Discrete Mathematics*, vol. 5, pp. 164–186, 1984.

[11] Y. Takahashi, "A lumping method for numerical calculations of stationary distribution of Markov chains," B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, 1975.

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

[13] H. Vantilborgh, "Aggregation with an error $O(\epsilon^2)$," *Journal of the ACM*, vol. 32, pp. 162–190, January 1985.

[14] D. Gross, B. Gu, and R. M. Soland, "Iterative solution methods for obtaining the steady-state probability distributions of Markovian multi-echelon repairable item inventory systems," *Computers and Operations Research*, vol. 20, pp. 817–628, October 1993.

[15] G. Franceschinis and R. Muntz, "Bounds for quasi-lumpable Markov chains," in *Performance Evaluation*, vol. 20, pp. 223–243, 1994.

[16] G. Franceschinis and R. Muntz, "Computing bounds for the performance indices of quasi-lumpable well-formed nets," *IEEE Transactions on Software Engineering*, vol. 20, pp. 516–525, July 1994.

[17] L. M. Malhis, *Development and application of an efficient method for the solution of stochastic activity networks with deterministic activities*. PhD thesis, University of Arizona, Tucson, Arizona, 1996.

[18] W. H. Sanders, W. D. Obal, M. A. Qureshi, and F. K. Widjanarko, "*UlltraSAN* modeling environment," *Performance Evaluation Journal*, vol. 24, pp. 89–115, October-November 1995.

[19] E. de Souza e Silva, H. R. Gail, and R. R. Muntz, "Efficient solution for a class of non-Markovian models," in *Proceedings of the Second International Workshop on the Numerical Solution of Markov Chains*, (Raleigh, North Carolina), pp. 483–506, January 1995.