A Multi-Level Solution Algorithm for Steady-State Markov Chains

Graham Horton *
Lehrstuhl für Rechnerstrukturen, Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Federal Republic of Germany
graham@immd3.informatik.uni-erlangen.de

Scott T. Leutenegger †
Institute for Computer Applications in Science and Engineering
Mail Stop 132c, NASA Langley Research Center, Hampton, VA 23681-0001
leut@icase.edu

Abstract

A new iterative algorithm, the *multi-level* algorithm, for the numerical solution of steady state Markov chains is presented. The method utilizes a set of recursively coarsened representations of the original system to achieve accelerated convergence. It is motivated by multigrid methods, which are widely used for fast solution of partial differential equations. Initial results of numerical experiments are reported, showing significant reductions in computation time, often an order of magnitude or more, relative to the Gauss-Seidel and optimal SOR algorithms for a variety of test problems. It is shown how the well-known iterative aggregation-disaggregation algorithm of Takahashi can be interpreted as a special case of the new method.

1. Introduction

Markov systems generated by computer modeling tools such as queueing networks, Petri nets, or reliability modeling packages may contain hundreds of thousands of states. The resulting sparse linear systems of equations have a correspondingly large number of unknowns and must, in general, be solved numerically using an iterative scheme. Typical methods are the Power, Gauss-Seidel, and SOR methods. All of these methods have the drawback that they may require many iterations to reach a solution, particularly if the system is large, or a if high degree of accuracy is required. This can lead to unacceptably long computation times.

A similar situation is found when solving partial differential equations, where systems of many hundreds of thousands of unknowns are not uncommon. Here, however, a relatively new algorithm - the multigrid method - has met with considerable success, achieving, under appropriate conditions, substantially improved solution speeds compared to traditional methods such as Gauss-Seidel and SOR. One should more accurately consider multigrid to be a class of methods, as the basic framework allows a wide variety of choices for each of the constituent components. The method to be presented in this paper is in many respects related to this class of algorithms: the Markov system is recursively coarsened and values obtained from these smaller systems are used to achieve faster convergence.

In this paper we present a new solution algorithm, the *Multi-Level* algorithm, for Markov chains. Our initial experiments indicate the multi-level algorithm does not require the Markov chain to have any special structure in order to achieve excellent performance, although if such structure does exist it may be possible to exploit that structure to achieve even better results. We can offer no proof of convergence, but in all experiments we have run so far the method always converged. The convergence theory for multigrid methods is relatively limited, applying largely to equations of elliptic type. However the methods are widely used on all classes of linear and non-linear PDEs. We present experimental results for the Gauss-Seidel, SOR, and Multi-Level algorithms when applied to Markov chains generated from birth-death processes, finite population tandem queueing networks, blocking (finite capacity) tandem queueing networks, and a canonical stochastic Petri-net model.

For purposes of brevity we will refer to the Gauss-Seidel algorithm as the GS algorithm, and the Multi-Level algorithm as the ML algorithm. Note that the phrase "multi-level algorithm" is also used to denote a class of methods related to multigrid. These bear a structural resemblance to the scheme presented here in that they make use of coarser subproblems to achieve accelerated solution; they are, however, otherwise unrelated. The remainder of the paper is structured as follows. In the following section, after some preliminary remarks, the multi-level method is described. In section 3 we describe related work and compare and contrast the multi-level algorithm with existing algorithms. In section 4 results of experiments are presented comparing the performance of the method to GS and SOR using a variety of test problems. In section 5 we discuss

*This work was carried out in part while the first author was a guest at ICASE, NASA Langley Research Center.

†This research was supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the second author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

the practical aspects of implementation of the multi-level method and provide a list of possible directions for future research. In the final section we summarize the paper.

2. Multi-Level Solution Algorithm

In this section we present our new ML solution algorithm. We first summarize classical multigrid techniques in section 2.1. We then review classical Markov chain aggregation techniques in section 2.2. In section 2.3 we give a detailed description of our ML algorithm.

2.1 Multigrid Methods

Multigrid algorithms are a recent development in the field of iterative solvers for large systems of equations [1]. They were originally applied to the systems of equations that arise from the discretization of elliptic boundary value problems, and it is for these equations that most multigrid theory has been developed. For this class of problems multigrid algorithms are among the fastest known solvers, being of optimal complexity, i.e. having computation times that are linear in the size of the input. An introduction to multigrid algorithms may be found in [2], [7] or [18].

Multigrid algorithms begin by defining a set of increasingly coarse representations (grid levels) of the original problem, each of which has only a fraction of the number of degrees of freedom as its predecessor. The algorithm uses a standard iterative procedure such as GS at each grid level to quickly reduce error components that are high frequency w.r.t. that level (*smoothing*). A smoothing sweep through all grid levels efficiently reduces errors across the entire frequency spectrum.

Multigrid algorithms work most efficiently on regularly structured elliptic problems, whose coefficients vary smoothly between neighboring unknowns. In such cases they can achieve an error reduction of an order of magnitude per iteration. Conversely, multigrid algorithms for problems that are non-elliptic, unstructured or have rapidly varying coefficients do not in general perform as well and currently represent an active field of research. Markov chains can possess one or more of the above characteristics. The branch of multigrid research which attempts to deal with general sparse systems is known as *algebraic multigrid*, see [12].

Given an appropriate choice of smoother, multigrid algorithms can be parallelized with a high degree of efficiency, see [9] for a recent survey. This is, of course, an added advantage in the context of modern supercomputer architectures and networked workstations.

The algorithm to be presented in section 2.3 may be viewed as a multigrid-like algorithm. However, owing to the absence of a grid structure and because of the difference in approach to multigrid schemes, we will refer to the algorithm more abstractly as a *multi-level* algorithm. Because of the similarities between the algorithms, we nevertheless expect that many of the established multigrid ideas can be applied to the Markov chain solver, and this indeed proves to be the case.

2.2 Aggregation of Markov Chains

Consider a steady state continuous time Markov chain consisting of n states $s_1 \dots s_n$. Denote the unknown vector by p, where p_i is the probability of being in state s_i.

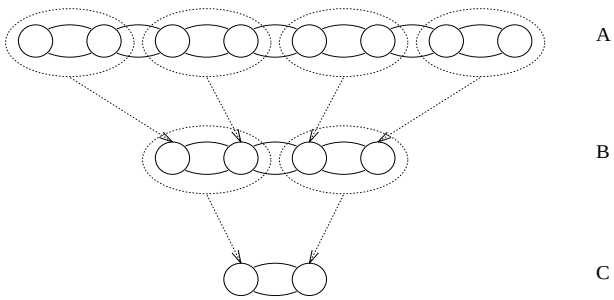


Figure 1: Aggregation of Markov Chains

We then have to solve the system of equations

$$Pp = 0 \tag{1}$$

with the additional condition

$$\sum_{i=1}^{i=n} p_i = 1 \tag{2}$$

Equation (1) is simply a reformulation of the classic continuous time Markov chain equation:

$$\pi Q = 0 \tag{3}$$

where P is the transpose of the generator matrix Q, and p is the transpose of the steady state probability row vector π. Note that we will use the symbol Q differently in this paper. Equations (1) and (2) form a sparse linear system which is typically solved numerically using the GS or SOR algorithm. These schemes suffer the drawback of needing a large number of iterations when n is large or when a high degree of accuracy is required.

A coarser representation of the Markov chain described by matrix P may be obtained by *aggregation*. This means creating a new Markov chain described by a matrix Q with the vector of state probabilities q, each of whose N states $S_1 \dots S_N$ is derived from a small number of states of the original system. Figure 1 illustrates the situation for an eight-state birth/death chain (A), where states are aggregated in pairs to form a four-state coarser level system (B), which in turn is pairwise aggregated to form the coarsest level two-state system (C).

In the following we will use the terms *fine level* and *coarse level* to refer to Markov chains where the latter is obtained by aggregation from the former. The relation $s_k \in S_i$ signifies that the fine level state s_k is mapped by the aggregation operation to the coarse level state S_i.

The matrix Q of the aggregated system may be chosen as follows :

$$Q_{ji} = \frac{\sum_{s_k \in S_i} p_k \sum_{s_l \in S_j} P_{lk}}{\sum_{s_k \in S_i} p_k} \tag{4}$$

This is the classical *aggregation equation*. Note that the matrix Q is a function not only of the fine level matrix P, but also of the fine level solution vector p.

This yields the aggregated equations in the unknown q:

$$Qq = 0 \quad , \tag{5}$$

$$\sum_{i=1}^{N} q_i = 1 \quad . \tag{6}$$

It can then be shown that

$$q_i = \sum_{s_k \in S_i} p_k \quad , \tag{7}$$

i.e. the solution q of the aggregated system truly represents a coarser version of the solution p of the original problem. The probability of being in state q_i is the sum of the probabilities of being in any of its constituent fine-level states.

The idea behind the ML algorithm is to aggregate slowly, typically setting $N = n/2$ or $N = n/4$, and to proceed recursively to obtain further coarsenings, arriving eventually at some coarsest system which may consist of only two states. Approximations obtained on coarser systems are used to obtain a *correction* to the fine-level solution vector. One iteration of the ML algorithm will proceed in the multigrid manner from the original, finest system down to the coarsest, setting up coarse level equations and performing GS smoothing, and then back up to the finest level, computing and applying corrections. Note, however, that other orderings of processing the various levels are possible.

2.3 Description of the Algorithm

We adopt the following abbreviations for vectors $a,\ b,\ c \in \Re^m$:

$$\begin{aligned} a = b * c \quad &\equiv \quad a_i = b_i * c_i, \quad 1 \le i \le m \\ a = b/c \quad &\equiv \quad a_i = b_i / c_i, \quad 1 \le i \le m \end{aligned}$$

We enter the $(i+1)$th iteration of the ML algorithm with the current approximation to the solution $p^{(i)}$ obtained as a result of the (i)th iteration, whereby $p^{(0)}$ denotes the initial guess, and begin by performing one or more sweeps of the GS algorithm, obtaining the vector $\tilde{p}$:

$$\tilde{p} = GS(p^{(i)}) \quad . \tag{8}$$

We assume throughout that application of GS includes a subsequent normalisation step to enforce (2). The vector $\tilde{p}$ will not, in general be the solution p of (1), but we may write

$$\tilde{p} * p^* = p \quad , \tag{9}$$

where p^* is the *elementwise multiplicative correction* necessary to $\tilde{p}$. Knowledge of p^* would immediately enable us to compute the solution p. We may write (1) as

$$P(\tilde{p} * p^*) = 0 \quad . \tag{10}$$

Since $\tilde{p}$ has been smoothed by application of the GS algorithm, we assume that it no longer contains any high frequency error components, and thus that p^* is smooth. Therefore we may compute an approximation to p^* on a coarsened system, since the dimension of the latter is smaller, and thus the computation will be cheaper. We write a coarsened version of (10) as

$$\tilde{Q}(\tilde{q} * q^*) = 0 \quad , \tag{11}$$

where $\tilde{Q}$ is the matrix of the aggregated system and q^* and $\tilde{q}$ represent aggregated representations of p^* and $\tilde{p}$, respectively.

The coarse system matrix $\tilde{Q}$ is chosen to be an approximation to the the matrix Q from (4), replacing p by $\tilde{p}$, since p is not available until the algorithm has converged:

$$\tilde{Q}_{ji} = \frac{\sum\limits_{s_k \in S_i} \tilde{p}_k \sum\limits_{s_l \in S_j} P_{lk}}{\sum\limits_{s_k \in S_i} \tilde{p}_k} \quad . \tag{12}$$

In the case of a converged solution, we will, however, have $\tilde{p} = p$ and therefore the correct coarse matrix $\tilde{Q} = Q$.

In order to obtain $\tilde{q}$ in (11) we require an operator that maps a fine level vector to the coarser, aggregated vector. This operator will be denoted by R (from the multigrid restriction operation), and we write

$$\tilde{q} = R(\tilde{p}) \quad . \tag{13}$$

We choose summation for R, in accordance with (7):

$$\tilde{q} = R(\tilde{p}) \quad \equiv \quad \tilde{q}_i = \sum_{s_k \in S_i} \tilde{p}_k \quad . \tag{14}$$

This choice for R has the property

$$q = R(p) \quad ,$$

i.e. the exact fine level solution is mapped by R to the exact coarse level solution. It is clear that this property is necessary, since at convergence, both (1) and (5) must be satisfied.

We proceed by defining

$$\bar{q} = \tilde{q} * q^* \quad , \tag{15}$$

thus obtaining the coarse level equations to be solved:

$$\tilde{Q}\bar{q} = 0 \ , \quad \sum_{i=1}^{N} \bar{q}_i = 1 \quad . \tag{16}$$

Solving (16) for $\bar{q}$ will therefore enable us to compute q^*, the coarse approximation to the correction via (15): $q^* = \bar{q}/\tilde{q}$.

We compute the fine-level correction from its coarse approximation using the operator I (interpolation):

$$p^* = I(q^*) \quad . \tag{17}$$

We choose the following operation for I:

$$p^* = I(q^*) \quad \equiv \quad p^*_k = q^*_i \quad s_k \in S_i \quad . \tag{18}$$

The multiplicative correction is equivalent to a scaling: a coarse level node value $\bar{q}_i$ represents the probability of being in any state $s_k \in S_i$ and therefore q^* represents the scaling factor necessary to achieve this for the values $p_k,\ s_k \in S_i$. In

this respect the ML algorithm differs from multigrid, where an additive correction is performed.

We then compute the new iterate $p^{(i+1)}$ using

$$p^{(i+1)} = \bar{p} = C(\tilde{p}, p^*) \quad \equiv \quad \tilde{p} * p^* \quad . \tag{19}$$

If the algorithm converges, we hope that $p^{(i+1)}$ will be a better approximation to p than $p^{(i)}$. Meanwhile, however, we have $p^{(i+1)} \neq p$, since the correction p^* was computed only approximately on a coarse grid, using an incorrect matrix $\tilde{Q} \neq Q$.

By analogy with the SOR scheme, which computes an over-corrected iterate compared to the underlying GS algorithm, we may consider using an overcorrection for the ML scheme:

$$p^* = I(q^*) \quad \equiv \quad p_k^* = q_i^*(\omega + (1-\omega)q_i^*) \quad s_k \in S_i \quad , \tag{20}$$

where we set $0 \leq \omega \leq 1$. For $\omega < 1$ such an operation will overdo the correction, since values $q_i^* < 1$ will be decreased and values of $q_i^* > 1$ will be enlarged. The parameter ω thus plays an analogous role to that of the over-relaxation parameter in the SOR scheme. It is to be hoped that, as is the case for the SOR scheme, certain choices of ω may lead to improved solution efficiency. We will consider the utility of over-correction in section 4.2.

After the iteration has converged, we have

$$q^* = \mathbf{1}^N \quad , \quad p^* = \mathbf{1}^n \quad ,$$

where $\mathbf{1}$ denotes the vector $(1,\ 1,\ \ldots,\ 1)^T$, i.e. no further correction takes place. We then also have $\tilde{Q} = Q$ and therefore $\bar{q} = q$.

Note that the question of assigning fine nodes to their coarse counterparts is still open. This we will call the *coarsening strategy* or *aggregation strategy*. The aggregation strategy can have a significant effect on the performance of the ML algorithm. In cases where we have some knowledge of the structure of the Markov chain, for example when queueing networks are to be modelled, then we may utilize this information in the construction of the aggregated system. In other cases, mapping strongly coupled fine states to the same aggregated state seems to be an efficient strategy.

Note that the composite coarse grid correction operator $C(\tilde{p}, I(q^*))$ preserves the *relative probabilities* of all fine level states mapped to the same coarse level state by aggregation:

$$\bar{p} = C(\tilde{p}, I(q^*)) \Rightarrow \frac{\bar{p}_{k_1}}{\bar{p}_{k_2}} = \frac{\tilde{p}_{k_1}}{\tilde{p}_{k_2}} \quad s_{k_1}, s_{k_2} \in S_i, \quad 1 \leq i \leq N \quad .$$

The *two*-level version of the ML iteration is given by the sequence of steps (8), (13), (16), (17), (19). The *multi*-level algorithm is obtained by recursive application of the two-level algorithm to obtain a solution to the aggregated equation (16) and is described in algorithmic form in Figure 2. We use the subscript l to denote level of representation ($l = lmax$ finest level, $l = 0$ coarsest level). The coarse level $l-1$ and fine level l between which the operators I and R map are identified by appropriate indices. Note that, because of the recursive nature of the algorithm, the unknowns q^*, $\bar{q}$ and $\tilde{q}$ are represented by the variables p_{l-1}^*, $\bar{p}_{l-1}$ and $\tilde{p}_{l-1}$, respectively.

```
procedure mss(l)
   if (l = 0)
      solve P_l p̄_l = 0
   else
      p̃_l = GS^ν(p̄_l)
      p̃_{l-1} = R_{l-1,l}(p̃_l)
      mss(l − 1)
      p*_{l-1} = p̄_{l-1} / p̃_{l-1}
      p*_l = I_{l-1,l}(p*_{l-1})
      p̄_l = C(p̃_l, p*_l)
   return
```

Figure 2: Multi-Level Algorithm

The Multi-Level algorithm is non-linear, owing to the use of the coarsened system obtained via (12), although the original problem (1) is linear. It seems therefore unlikely that theoretical results can be obtained for estimates of the convergence speed of the algorithm for general problems.

We allow in general the possibility of applying GS ν times at each level with $\nu \geq 1$, denoted by $\mathtt{GS}^\nu$. We also consider the possibility of a more complex cycle form (in particular F- and W-cycles, see the multigrid literature), obtained by multiple recursive calls to procedure `mss`.

3. Related Work

Currently most performance tools requiring the solution of Markov chains use either the power, GS, or SOR algorithms. In a paper by Stewart and Goyal [16] the various techniques are compared and SOR with dynamic tuning of the relaxation parameter emerges as the method of choice. Initial results of the ML algorithm show that it generally outperforms optimal SOR, often by on order of magnitude or more, without any parameters to tune.

Our work is related to a large body of work on aggregation-disaggregation techniques. Most previous work using aggregation makes the assumption that the number of aggregate states, N, is much less than the number of states, n, i.e. $N \ll n$. In our algorithm we generally assume $N = \frac{n}{2}$.

In addition, much of the related work assumes that the Markov chains being solved are generalized birth-death processes [17, 15], or that the Markov chains are nearly completely decomposable systems [5, 6] In the latter case the solution is usually an approximation often accompanied by bounds on the error. We refer the reader to [13] for descriptions of these special Markov chain structures and for a more comprehensive list of references. Our work differs in that it does not require any special structure in the Markov chain, and the result is exact, not an approximation.

The work that most strongly resemble ours is the algorithm of Takahashi [17] and its variants [13]. We subsequently use the terminology derived in the previous section. The Takahashi algorithm starts with an initial

iterate for the fine level chain. The fine level chain of n states is then aggregated into a coarse Markov chain of N states, where $N \ll n$ using equation (12). The coarse chain is then solved (equation (16)), and the correction obtained from the coarse chain is applied to the previous iterate in the fine level. A new iterate at the fine level is obtained as follows. The fine level states are grouped together into N sets of states where the members of each set correspond to each aggregate state in the coarse level. The set of linear equations corresponding to each of these sets of states is solved independently of the other fine level states by treating the values for the other states as constants. The new iterate at the fine level is the result of solving the equations for each set of states. The algorithm iterates between the two levels until sufficient accuracy is obtained.

We may view the Takahashi algorithm as a special case of ML, obtained by the following choices, compared to the ML scheme we prefer:

1. Use of only two levels of representation of the system, rather than multiply coarsened problems.
2. $N \ll n$, as opposed to $N = \frac{n}{c}$ for a small c (typically 2 or 4).
3. Use of Block Gauss-Seidel or Block Jacobi on the finer level, as opposed to a small number of steps of a pointwise Gauss-Seidel scheme.

Although the fundamental motivation for the Takahashi and the ML algorithms is similar: computation of local probability distribution on the fine level and achieving global probability redistribution using the coarsened equation, it is our claim that 1, 2 and 3 above are not the best choices. Point 1 above leads to a system of equations on the coarse level of size N, and point 3 generates N subsystems of size n/N, all of which have to be solved at each iteration, which can become prohibitively expensive. Point 2 above implies that each new coarse node value has to serve as a correction for a large number of fine states. By contrast, the ML algorithm never requires the solution of medium or large equations and provides corrections with a high ratio of coarse to fine values.

We restate that our motivation for the ML algorithm was not to modify current aggregation-disaggregation algorithms, but rather to devise a algorithm similar to multi-grid algorithms which have shown exceptional merit in solving elliptic PDEs. We find it helpful to not view our algorithm as a Markov chain aggregation-disaggregation variant, but instead view it as a multigrid-like scheme.

4. Experimental Results

In this section we present experimental results to show how our new algorithm compares with GS and SOR. All experiments presented assume continuous time Markov chains. We have also solved discrete time Markov chains with similar improvements in performance relative to SOR and GS. In section 4.1 we first compare ML to GS and SOR for a variety of test problems using an unsophisticated implementation of the ML algorithm. In section 4.2 we demonstrate the potential of improving ML performance via techniques including intelligent aggregation, varying the number of smoothing steps at each level and over-correction.

4.1 Generic Multi-Level Results

In this section we present our generic ML results. By generic we mean that the ML algorithm used is the simplest one possible, a V cycle (each iteration goes from the finest level down to the coarsest level and back up to the finest level), no overcorrection, only applying one iteration of the smoother (GS) at each level, and a simple aggregation strategy. In particular, we assume states of the Markov chain are ordered $0 \ldots n-1$, and that the aggregation policy is by pairing neighboring states by index: $s_i \in S_I \Leftrightarrow I = \left\lfloor \frac{i}{2} \right\rfloor$. If a level has an odd number of states then the last state is included in the last coarse level state. Unlike SOR, this ML algorithm has no parameters to tune. In all of our experiments we give the benefit of the doubt to SOR and assume we can find the optimal relaxation parameter ω. We find ω to the nearest $\frac{1}{1000}^{th}$ by using a binary search between 1.0 and 2.0. This results in over-optimistic metrics for the SOR algorithm since in practice ω must be found via dynamic tuning, thus resulting in additional iterations. By presenting best case results for SOR and worst case results for ML, we strengthen our argument that ML may be a more promising solution algorithm than SOR. In all experiments we set the initial iterate to the vector $(\frac{1}{n}, \ldots, \frac{1}{n})^T$ and the systems are solved with with all methods to an accuracy of $\|Pp^{(i)}\|_2 \leq 10^{-6}$. The ML algorithm recursively coarsens the set of equations until the coarsest sytem has only two states.

Possible metrics for comparing the algorithms are the number of iterations, the process time, the number of floating point operations (flops), and the geometric mean of the convergence rate. The number of flops was computed by inserting a counter into all three programs. The process time is obtained from the unix system call *times()* and is the CPU time used while executing instructions in the user space of the programs. In all cases we have found the process time to be a more conservative measure than flops, i.e. the comparison of ML to GS and SOR is less favorable when using process times than when using flops. Hence to strengthen our arguments of the utility of the ML algorithm we chose process time as our primary metric. Note that the process time also includes time for generation of the additional ML data structures, whereas the flops metric would miss this factor. In addition, the flops metric does not capture the additional pointer operations needed by ML for accessing elements in the coarse levels. We also consider the number of iterations necessary for convergence. In general, ML requires far fewer iterations than the other two algorithms, but consumes more time per iteration since each iteration requires application of the smoother at each level and the construction of the coarse level matrices $\tilde{Q}$.

We first consider a birth-death Markov chain with a birth rate of 1 and a death rate of 2 and a varying the number of states. The results are presented in figure 3. The number of iterations increases linearly with the number of states for the GS and SOR algorithms, whereas it remains fixed at 21 iterations for the ML algorithm. Intuitively, probability

mass only moves slowly through the system in the GS and SOR algorithms, whereas one iteration of the ML algorithm can move mass from one end of the chain to the other via the coarse level problems.

The process time of SOR and GS increases quadratically for SOR and GS with the number of states, whereas it increases only linearly for ML. Thus ML is an optimal method for this particular problem. The GS (SOR) algorithm requires 257 (128) times more processing time than ML for a birth-death chain of 10,000 states. The ratios increase with system size. Even for small birth death chains, such as 1,000 states, the ML algorithm is more than an order of magnitude faster than GS and SOR.

One possible reason that the GS and SOR algorithms require more time as the number of states is increased is that they must move probability mass a longer distance in the solution vector. To determine whether it is this distance or the overall number of states we conduct the following experiment. We assume a birth death chain with each state having two additional exiting transitions beyond the birth and death. State s_i has a transition to state s_{i+m} and a transition to state s_{i-m}. If $(i+m) > n-1$ then the transition is to state $n-1$. Similarly, if $(i-m) < 0$ then the transition is to state s_0. We initially set the number of states equal to 500 and m equal to 2. We then successively double the number of states and the distance m. We define the *diameter* of a Markov chain to be the maximum distance (or number of transitions) between any two states. Hence, in this experiment, regardless of the number of states, the diameter is fixed at 250. We assume all transitions in the birth direction proceed at rate 1, and all transitions in the death direction are at rate 2.

The results from this experiment are presented in figure 4. It appears that both the size *and* diameter of the Markov chain influence convergence speed of GS and SOR. The number of iterations for the ML algorithm varies only between 19 and 25. Thus the diameter of the chain does not appear to have much effect on the solution speed (measured in iterations) of the ML algorithm.

We next investigate the sensitivity of the relative performance of the algorithms to the ratio of birth rate to death rate. We fix the birth rate at 1 and vary the death rate from 0.001 to 1000. The number of states is equal to 10,000. The results are presented in figure 5. Note that both axes are scaled logarithmically.

The performance of the GS algorithm is always worse than that of the ML algorithm. The SOR algorithm performs better than ML when the death rate is less than the birth rate, but one to two orders of magnitude worse when the roles are reversed. In the best case observed (for SOR), SOR requires $\frac{1}{4}$ of the process time of ML. Note that the computation times for GS and SOR could be made symmetric by exchanging the birth and death rates or by reordering the states. The ML algorithm does not require any such special techniques to achieve good performance, and hence is more resilient to changes in transition rates.

The excellent performance of SOR when the death rate is less than the birth rate is surprising. In fact, *it is not realistic*. For a ratio of $\frac{2}{3}$ or less SOR converges in less than 30 iterations. We were able to achieve this excellent performance by first determining the optimal ω to the nearest $\frac{1}{1000}^{th}$ by applying SOR many times with ω values chosen in a binary search. In practical situations the optimal value of ω is not known a priori and the solution will be calculated only once. Hence ω must be obtained via some dynamic procedure. It is impractical to assume that an SOR algorithm including dynamic tuning of ω can converge in only 30 iterations. In fact, in a recent paper proposing a dynamic method for determining ω, [3] , 30 iterations are executed before tuning of ω even begins. The paper notes that often thousands of iterations were necessary to find the optimal ω. Thus, the excellent performance of SOR in figure 5 could never be achieved in practice.

We next explore Markov chains generated from simple queueing models. We first assume a closed system tandem queue model. The queueing system is shown in figure 6. We assume a finite population where jobs think for an exponentially distributed period of time with rate λ (i.e. the jobs visit an infinite server and are served at rate λ). Jobs are served in queues 1 and 2 at rates μ_1 and μ_2 respectively. The states of the Markov chain are generated from an initial state of all jobs in the think state, and then by constructing the chain in a breadth-first fashion. States are numbered 0 through $n-1$ as they are created in the breadth first search. The aggregation policy used in the ML algorithm is to pair adjacent-numbered states. Hence, the performance of the ML algorithm is almost certainly sub-optimal, since no intelligent aggregation techniques are being used. In all experiments reported we fix the think rate to 1.0.

In the first experiment we set μ_1 to 1.0 and μ_2 to 2.0 and vary the population from 25 to 250. This results in a range of 351 to 31,626 states in the underlying Markov chain. The results of the experiment are plotted in figure 7. The ML algorithm requires far less computation time for the solution than the other algorithms, especially as the population increases.

We next consider the effect of the relative values of μ_1 and μ_2 on the performance of the algorithms. We fix the think rate to 1.0 and the population to 100 (resulting in 5151 states in the underlying Markov chain), set μ_1 to 1.0, and vary μ_2 from 0.001 to 1000. The results are plotted in figure 8 using a logarithmic scaling of both axes. The ML policy results in lower computation times than the other algorithms across the entire parameter space, and when $\mu_1 > \mu_2$ the solution time is an order of magnitude faster. Experiments with larger populations demonstrate a more pronounced difference in the performance of ML relative to GS and SOR.

We now consider the application of the three algorithms to the solution of the underlying Markov chain of a canonical stochastic Petri net. Figure 10 shows the complexities, measured in KFLOPs and plotted logarithmically, of the GS, SOR and ML algorithms applied to the underlying Markov chain of the stochastic Petri net of Molloy [10], depicted in Figure 9. In this experiment the number of tokens ranged from 10 to 60, resulting in Markov chains with 506 to 77531 states. The Markov chains for this

	$1-D$			$2-D$		
ν	1	2	3	1	2	3
$MFLOPs$	12.7	9.3	8.2	5.4	5.2	4.9

Table 1: Effect of ν and aggregation strategy on the ML algorithm.

problem were generated using the SPNP (*stochastic Petri net package*) tool of Ciardo et al [4]. ML is superior to GS for all cases tested, the improvement being a factor of approximately 3.3 for the smallest and approximately 42.7 for the largest case considered. Against SOR with an optimally chosen ω, ML performs slightly worse only for the smallest problem considered. A comparison of computation times shows similar, but slightly less good results, owing to an increased number of page faults caused by ML's greater memory requirements. The aggregation strategy used was the simplest possible, using only pairwise aggregation by index. Initial experiments with more a sophisticated scheme indicate that further improvements are possible.

4.2 Multi-Level Acceleration Techniques

In this section we consider a few techniques, some of which are borrowed from the multigrid literature, that can be applied to the ML algorithm to further accelerate convergence speed. The over-correction idea is directly analogous to over-relaxation in SOR.

We first consider how more intelligent aggregation of the Markov chain can affect the performance of the ML algorithm. We consider the same tandem queueing network in section 4.1, figure 6, except that we assume finite capacity (blocking) queues with a capacity of 63. We assume the queues are finite capacity to facilitate the ease of an intelligent aggregation technique. Figure 11 shows on the left the Markov chain generated by this modified tandem queue. The states of the Markov chain may be written as a two-dimensional lattice of size $(64) \times (64)$. The transitions then form a regular pattern, somewhat analogous to the grid of a discretized PDE. We assume the states to be numbered lexicographically from top left to bottom right. The simple aggregation strategy used would successively pair states that are adjacent horizontally until no longer possible and then pairwise vertically, as illustrated on the upper right. An alternative strategy more appropriate to the structure of the problem is shown on the lower right, where fine level states are grouped into 2×2 units.

Table 1 shows the number of floating point operations needed by ML applied to the Markov chain of Figure 11. We compare the one-dimensional, pairwise aggregation strategy (*1-D*) with the two-dimensional case (*2-D*). We consider from one to three smoothing steps ($\nu = 1,\ 2,\ 3$). For comparison, GS requires 30.1 MFLOPs and optimal SOR requires 14.1 MFLOPs. It can be clearly seen that the two-dimensional aggregation strategy is significantly faster than the 1-D method. The former can be also be improved (by about 33%) by performing additional relaxation steps, whereas the latter algorithm, which is already superior, improves to a lesser extent. In this experiment the best-case ML scheme achieves a speedup of 6.1 over GS and 2.9 over SOR.

ω	1.0	0.8	0.6	0.4	0.2	0.0
$MFLOPs$	5.38	4.67	3.78	3.25	3.07	3.07

Table 2: Effect of overcorrection on operation count.

Table 2 shows the effect of the "overcorrection" according to (20) on the operation count of the ML scheme applied to the same problem, where ω ranges from 1.0 to 0.0. A significant improvement is found to be achievable, the optimal value of ω giving rise to an improvement of approximately 53% over the unmodified correction. The ML scheme with the optimal overcorrection and adapted aggregation strategy thus requires less than 1/10 (1/5) of the floating point operations of the GS (SOR) algorithm.

5. Discussion of the algorithm and the results

Memory requirements. The implementation of the ML algorithm requires one additional variable at each state compared to the SOR scheme in order to store the temporary values $\tilde{p}$. In addition, the overall number of states needed is higher than for SOR because of the additional recursively aggregated systems. If the number of states of the original problem is n and this number is reduced by a factor of $f = N/n$ during each aggregation step, then the overall number of states s needed by the ML algorithm is bounded by $s < n/(1-f)$. Thus in the examples considered in section 4 we have $f = 1/2$ and therefore $s < 2n$. We therefore pay the price of circa three times the memory requirements of GS in order to achieve performance improvements of an order of magnitude or more.

Implementation effort. The ML algorithm is evidently more complex than the SOR scheme, additional coding being required for the the treatment of the coarse level equations. The implementations used in section 4 required however, only 329 and 576 lines of C for the SOR and the ML algorithms respectively. Thus we consider implementation overhead not to be significant.

Parallelization. The Multi-Level algorithm will parallelize well, given an appropriate choice of smoother. There are difficulties involved with the parallel execution of the GS and SOR algorithms, owing to their recursive structure. However, it has been shown that the "multi-color" style of GS can allow efficient parallelization without compromising convergence speed [9]. Parallelization of ML is done by data partitioning within each level. With a multi-color smoother, all the operations of the ML method at a given level can be performed concurrently. Communication will be required between processors for the smoothing step and collect/broadcast operations for the convergence test and enforcement of (2). Although coarser levels will run less efficiently, as less computation is performed there, the coarse granularity of the finer levels, where most computational work is located, will ensure overall good parallel performance. Thus we conclude that the ML algorithm, too,

will perform well on a multiprocessor system or workstation cluster.

Cycle types. There are other alternatives to processing the levels of aggregation in the downward-upward sweep used in the present scheme. These can be obtained by modifying the number of recursive calls to procedure `mss` in Figure 2. Thus coarser levels may be visited more than once during one iteration. Such techniques can, in the multigrid context, lead to improved efficiencies, as the coarse level equations are then more accurately solved. Experiments reported in [8] have shown that this can also be the case for ML. One may furthermore consider dynamic cycling after Brandt [1] or adaptive cycling according to Rüde [11].

Choice of coarsening strategy. Convergence characteristics may be improved by a judicious choice of aggregation strategy. In particular, Markov chains derived from queueing networks can possess a regular structure which may be exploited. Experiments have shown this to be the case for the tandem queue example of section 3. Our main priority for future work is therefore to develop an aggregation strategy which obtains appropriate coarse systems at a reasonable cost. Finding optimal aggregations is a notoriously complex problem, but we believe that sub-optimal solutions are sufficient for the ML algorithm.

6. Summary and Outlook

The ML algorithm presented in this paper has been shown to require significantly less computation time than the SOR scheme for a number of test problems. The difference between the two algorithms increases with the number of states of the Markov chain. In addition to being significantly faster, based on our experience to date it appears that the ML algorithm is much more resilient to variations in transition rates. Another nice property of the ML algorithm is that excellent performance can be obtained without the necessity of tuning a parameter, such as the over-relaxation parameter in SOR. The ML method settles quickly into a constant rate of convergence, implying that the computational work increases linearly with the accuracy requiremnts.

Examination of a larger sample of cases, including Markov chains from real applications have yet to be made. However, we feel that the algorithm shows enough promise to justify further investigation into ML schemes for solving Markov chains.

Further work will also include the implementation and testing of the techniques mentioned in the previous section. In particular, attention will be paid to choosing an aggregation strategy for general Markov chains, where no *a priori* knowledge of the topology is available. Several of these techniques have already proven to provide improved efficiency in preliminary experiments. We hope to be able to report on this in the near future.

An experimental comparison with the Takahashi algorithm is also planned.

A parallel version of the ML algorithm is also in preparation, and we hope to be able to present results obtained on a MIMD supercomputer in the near future.

Acknowledgements

The authors would like to thank J. Van Rosendale, D. Nicol, P. Heidelberger, and G. Ciardo for helpful discussions and suggestions for improvements and G. Ciardo for permission to use the SPNP tool.

References

[1] A. Brandt: *Multi-level adaptive solutions to boundary-value problems.* Math. Comp. 31, pp. 333-390, 1977.

[2] W. Briggs: *A Multigrid Tutorial.* SIAM, Philadelphia, PA, 1987.

[3] G. Ciardo, A. Blakemore, P. Chimento, J. Muppala, K. Trivedi: *Automated generation and analysis of Markov reward models using stochastic reward nets.* To appear in C. Meyer and R. Plemmons (Ed.) *Linear Algebra, Markov Chains, and Queueing Models*, IMA Volumes in Mathematics and its Applications, springer-Verlag, 1993.

[4] G. Ciardo, K. Trivedi, J. Muppala: *SPNP: stochastic Petri net package.* Proc. of the Third Int. Workshop on Petri Nets and Performance Models (PNPM89), Kyoto, Japan, pp. 142-151 Dec, 1989. IEEE Computer Society Press.

[5] P. Courtois: *Block Decomposition and Iteration in Stochastic Matrices.* Philips Journal of Research 39, 1984.

[6] P. Courtois, P. Semal: *Computable Bounds for Conditional Steady State Probabilities in Large Markov Chains ans Queueing Models.* IEEE Journal on Selected Areas in Communications, SAC-4, No. 6, 1986.

[7] W. Hackbusch: *Multi-Grid Methods and Applications*, Springer Verlag, Berlin, 1985.

[8] G. Horton, S. Leutenegger: *A Multilevel Solution Algorithm for Steady-State Markov Chains.* ICASE Report #93-81 NASA CR-191558, NASA Langley Resarch Center, September 1993.

[9] O. McBryan, P. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C. Thole, and U. Trottenberg.: *Multigrid methods on parallel computers — a survey of recent developments.* IMPACT of Computing in Science and Engineering, 3:1–75, 1991.

[10] M. Molloy: *Performance analysis using stochastic Petri nets.* IEEE Trans. Comp. Vol 31 No. 9, 913-917, Sept. 1982.

[11] U. Rüde: *On the Multilevel Adaptive Iterative Method.* Technical Report TUM-I9216, Computer Science Dept., Technische Universität München, 1992.

[12] J. Ruge, K. Stüben: *Algebraic Multigrid* in S. McCormick (Ed.): *Multigrid Methods*, SIAM, Philadelphia, 1987.

[13] P. Schweitzer: *A Survey of AggregationDisaggregation in Large Markov Chains.* In W. Stewart (Ed.) *Numerical Solution of Markov Chains*, Marcel Dekker, 1991, ISBN 0-8247-8405-7.

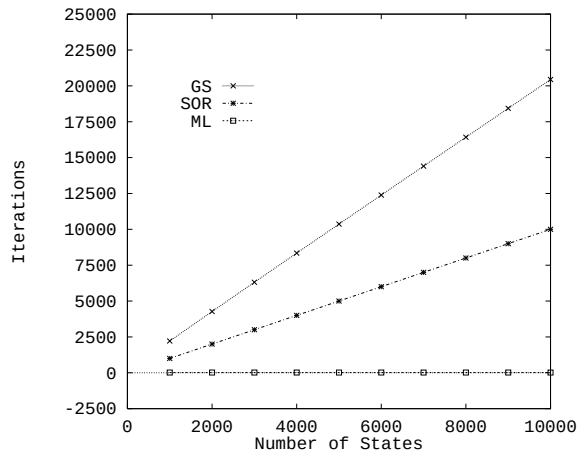
[14] P. Schweitzer: *Aggregation Methods for Large Markov Chains.* G. Iazeolla, P. Courtois, A. Hordijk (eds.): *Mathematical Computer Performance and Reliability.* Elsevier, 1984.

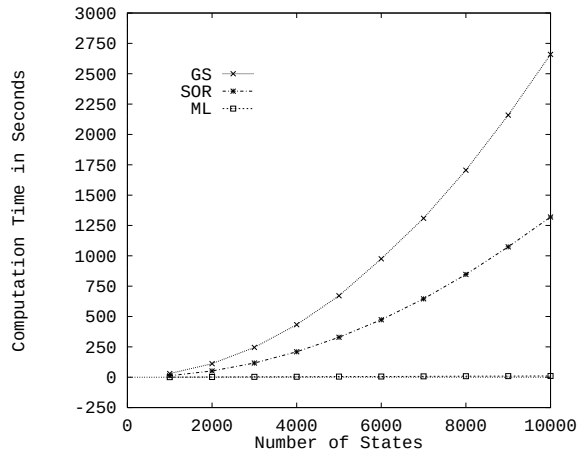
[15] L. P. Seelen: *An Algorithm for Ph/Ph/c Queues,* European Journal of Operations Research, V 23, pp 118-127, 1986.

[16] W. Stewart, A. Goyal: *Matrix Methods in Large Dependability Models.* IBM Research Report RC 11485 (#51598) 11/4/85.

[17] Y. Takahashi: *A Lumping Method for Numerical Calculations of Stationary Distributions of Markov Chains,* Research Report No. B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, 1975.

[18] P. Wesseling: *An introduction to multigrid methods*, John Wiley & Sons Ltd., Chichester, England, 1992.





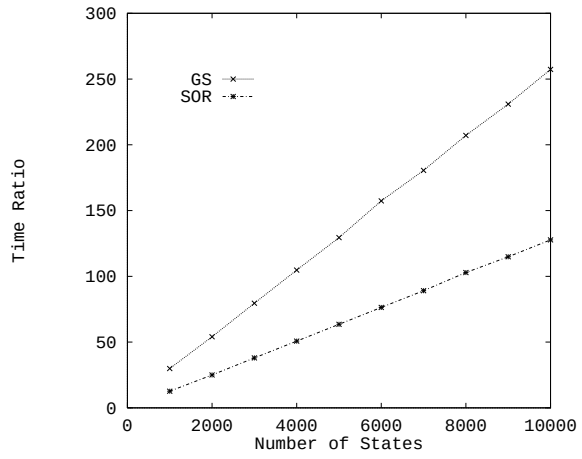


Figure 3: Birth-Death chain. Top : Number of Iterations; Center : Computation time; Bottom : Time Ratio.

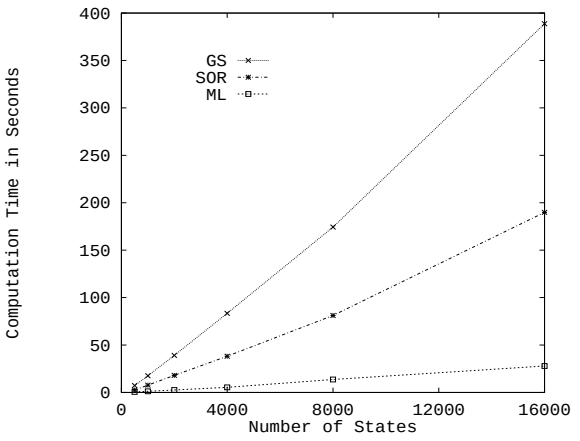
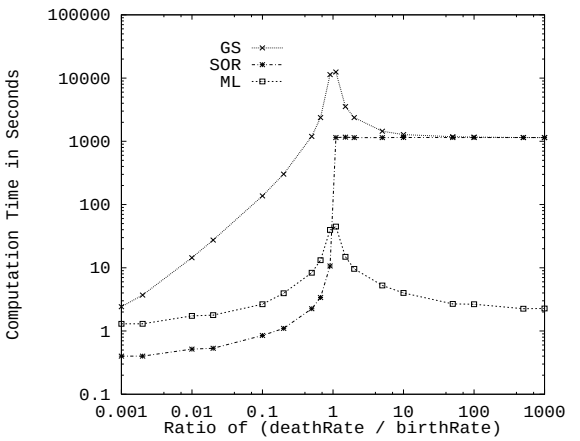


Figure 4: Birth Death problem with fixed diameter. Computation times.



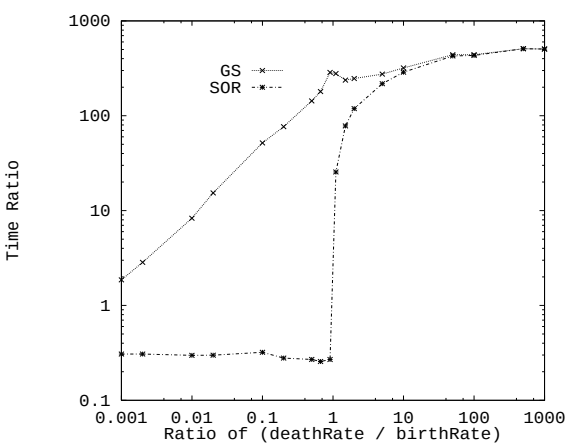


Figure 5: Birth-Death chain, effect of varying rates. Above : Computation time; Below : Time Ratio.

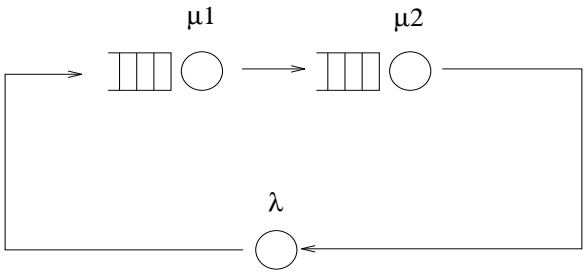


Figure 6: Tandem Queueing Network

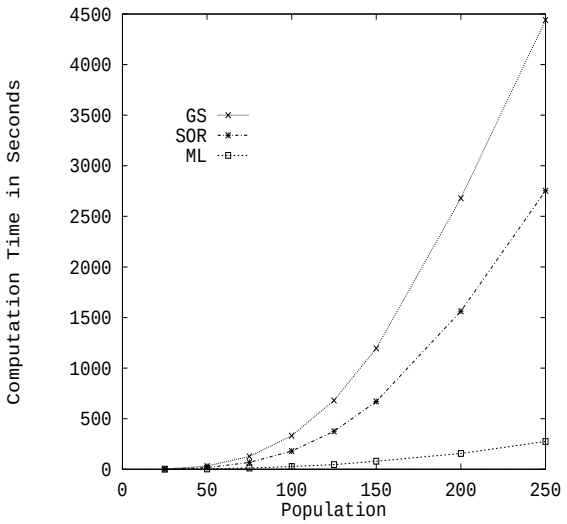


Figure 7: Tandem queue, variation of *pop*. Computation times.

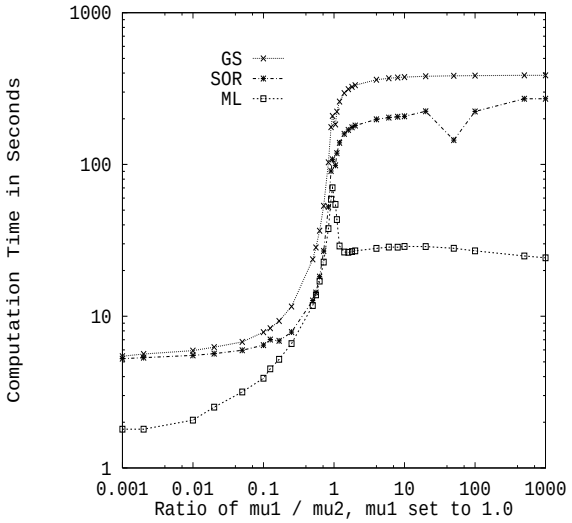


Figure 8: Tandem queue problem, variation of $\mu 1/\mu 2$. Computation time.

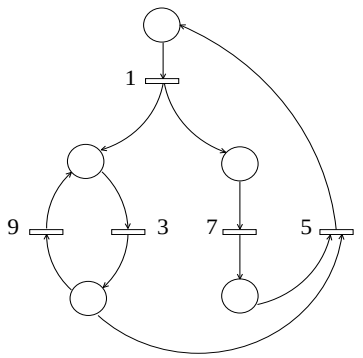


Figure 9: Molloy's Problem

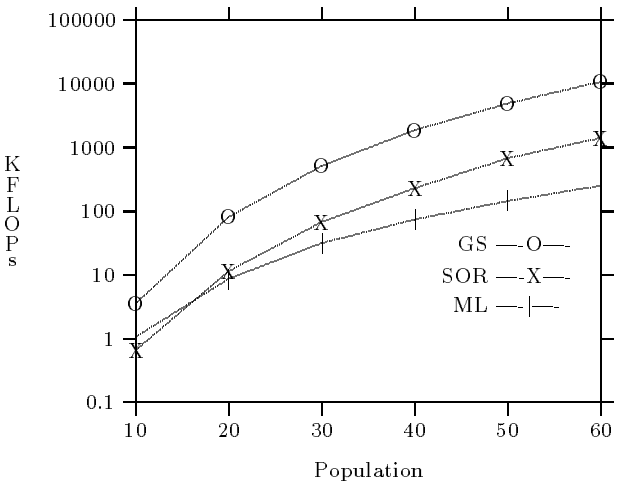


Figure 10: Results for Molloy's problem

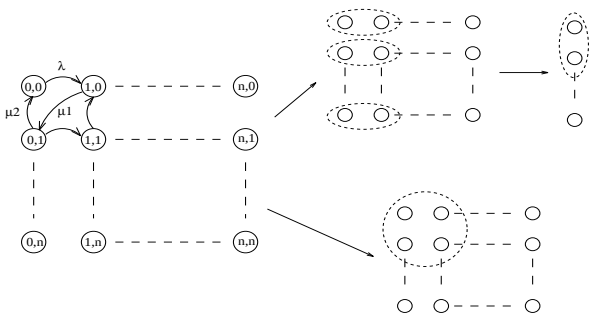


Figure 11: Markov state space for tandem blocking queue