Tackling Large State Spaces in Performance Modelling[⋆]

William J. Knottenbelt   Jeremy T. Bradley
{wjk,jb}@doc.ic.ac.uk

Department of Computing, Imperial College London, Huxley Building,
South Kensington, London SW7 2AZ, UK

Abstract. Stochastic performance models provide a powerful way of capturing and analysing the behaviour of complex concurrent systems. Traditionally, performance measures for these models are derived by generating and then analysing a (semi-)Markov chain corresponding to the model's behaviour at the state-transition level. However, and especially when analysing industrial-scale systems, workstation memory and compute power is often overwhelmed by the sheer number of states.
This chapter explores an array of techniques for analysing stochastic performance models with large state spaces. We concentrate on explicit techniques suitable for unstructured state spaces and show how memory and run time requirements can be reduced using a combination of probabilistic algorithms, disk-based solution techniques and communication-efficient parallelism based on hypergraph-partitioning. We apply these methods to different kinds of performance analysis, including steady-state and passage-time analysis, and demonstrate them on case study examples.

1 Introduction and Context

Modern computer and communication systems are increasingly complex. Whereas in the past systems were usually controlled by a single program running on a single machine with a single flow of control, recent years have seen the rise of technologies such as multi-threading, parallel and distributed computing and advanced communication networks. The result is that modern systems are complex webs of cooperating subsystems with many possible interactions.

In the face of this complexity, it is an extremely challenging task for system designers to guarantee satisfactory system operation in terms of both correctness and performance. Unfortunately, attempts to predict dynamic behaviour using intuition or "rules of thumb" are doomed to failure because designers cannot foresee the many millions of possible interactions between components. Likewise, *ad*

[⋆] Based on work carried out in collaboration with Nicholas J. Dingle, Peter G. Harrison and Aleksandar Trifunovic *hoc* testing cannot expose a sufficient number of execution paths. Consequently the likelihood of problems caused by subtle bugs such as race conditions is high.

One way to meet the above challenge using a rigorous engineering approach is to use formal modelling techniques to mechanically verify correctness and performance properties. The advantage of this style of approach over *ad hoc* methods has been clearly demonstrated in recent work on the formal model checking of file system code – in [1] the authors use a breadth-first state space exploration of all possible execution paths and failure points to automatically uncover several (serious and hitherto undiscovered) errors in ten widely-used file systems.

Formal techniques which consider all possible system behaviours can likewise be brought to bear on the problem which is the primary concern of the present chapter, namely that of predicting system performance. Our specific focus is on analytical performance modelling techniques which make use of Markov and semi-Markov chains to model the low-level stochastic behaviour of a system. (Semi-)Markov chains are limited to describing systems that have discrete states and which satisfy the property that the future behaviour of the system depends only on the current state. Despite these limitations, they are flexible enough to model many phenomena found in complex concurrent systems such as blocking, synchronisation, preemption, state-dependent routing and complex traffic arrival processes. In addition, tedious manual enumeration of all possible system states is not necessary. Instead, chains can be automatically derived from several widely-used high level modelling formalisms such as Stochastic Petri Nets and Stochastic Process Algebras.

A major difficulty often encountered with this approach is the *state space explosion problem* whereby workstation memory and compute power are overwhelmed by the sheer number of states that emerge from complex models. Consequently, a major challenge and focus of research is the development of methods and data structures which minimise the memory and runtime required to generate and solve very large (semi-)Markov chains. One approach to this “largeness” problem is to restrict the structure of models that can be analysed. This allows for the application of efficient techniques which exploit the restricted structure. Since these techniques are covered in other chapters, we do not discuss them further here, preferring unrestricted scalable parallel and distributed algorithms which are able to efficiently leverage the compute power, memory and disk space of several processors.

2 Stochastic Processes

At the lowest level, the performance modelling of a system can be accomplished by identifying all possible configurations, or *states*, that the system can enter and describing the ways in which the system can move between those states. This is termed the *state-transition* level behaviour of the model, and the changes in state as time progresses describe a *stochastic process*. We focus on those stochastic processes which belong to the class known as *Markov processes*, specifically continuous-time Markov chains (CTMCs) and the more general semi-Markov processes (SMPs).

Consider a random variable $\mathcal{X}$ which takes on different values at different times t. The sequence of random variables $\mathcal{X}(t)$ is said to be a stochastic process. The different values which $\mathcal{X}(t)$ can take, describe the state space of the stochastic process.

A stochastic process can be classified by the nature of its state space and of its time parameter. If the values in the state space of $\mathcal{X}(t)$ are finite or countably infinite, then the stochastic process is said to have a *discrete state space* (and may also be referred to as a *chain*). Otherwise, the state space is said to be *continuous*. Similarly, if the times at which $\mathcal{X}(t)$ is observed are also countable, the process is said to be a *discrete-time* process. Otherwise, the process is said to be a *continuous-time* process. In this chapter, all stochastic processes considered have discrete and finite state spaces, and we focus mainly on those which evolve in continuous time.

Definition 1. *A* Markov *process is a stochastic process in which the* Markov property *holds. Given that $\mathcal{X}(t) = x_t$ indicates that the state of the process $\mathcal{X}(t)$ at time t is x_t, this property stipulates that:*

$$\begin{aligned}\mathbb{P}\big(\mathcal{X}(t) = x \mid \mathcal{X}(t_n) = x_n, \mathcal{X}(t_{n-1}) = x_{n-1}, \ldots, \mathcal{X}(t_0) = x_0\big)& \\ = \mathbb{P}\big(\mathcal{X}(t) = x \mid \mathcal{X}(t_n) = x_n\big)& \\ \textit{for } t > t_n > t_{n-1} > \ldots > t_0&\end{aligned}$$

That is, the future evolution of the system depends only on the current state and not on any prior states.

Definition 2. *A Markov process is said to be* homogeneous *if it is invariant to shifts in time:*

$$\mathbb{P}\big(\mathcal{X}(t+s) = x \mid \mathcal{X}(t_n+s) = x_n\big) = \mathbb{P}\big(\mathcal{X}(t) = x \mid \mathcal{X}(t_n) = x_n\big)$$

2.1 Continuous-time Markov Chains

There exists a family of Markov processes with discrete state spaces but whose transitions can occur at arbitrary points in time; we call these continuous-time Markov chains (CTMCs). An homogeneous N-state CTMC has state at time t denoted $\mathcal{X}(t)$. Its evolution is described by an $N \times N$ generator matrix $\mathbf{Q}$, where q_{ij} is the infinitesimal rate of moving from state i to state j ($i \neq j$), and $q_{ii} = -\sum_{j \neq i} q_{ij}$.

The Markov property imposes a *memoryless* restriction on the distribution of the sojourn times of states in a CTMC. The future evolution of the system therefore does not depend on the evolution of the system up until the current state, nor does it depend on how long the system has already been in the current state. This means that the sojourn time ν in any state must satisfy:

$$\mathbb{P}(\nu \geq s + t \mid \nu \geq t) = \mathbb{P}(\nu \geq s) \tag{1}$$

A consequence of Eq. (1) is that all sojourn times in a CTMC must be exponentially distributed (see [2] for a proof that this is the only continuous distribution function which satisfies this condition). The rate out of state i, and therefore the parameter of the sojourn time distribution, is μ_i and is equal to the sum of all rates out of state i, that is $\mu_i = -q_{ii}$. This means that the density function of the sojourn time in state i is $f_i(t) = \mu_i \, e^{-\mu_i t}$ and the average sojourn time in state i is μ_i^{-1}.

A concept that is fundamental to reasoning about the performance of a CTMC is that of its steady state distribution – that is the long-run average proportion of time that a system spends in each of its states.

Definition 3. *A Markov chain is said to be* irreducible *if every state communicates with every other state, i.e. if for every pair of states i and j there is a path from state i to j and vice versa.*

Definition 4. *The* steady-state probability distribution $\{\pi_j\}$ *of an irreducible, homogeneous CTMC is given by:*

$$\pi_j = \lim_{t \to \infty} \mathbb{P}(\mathcal{X}(t) = j \mid \mathcal{X}(0) = i)$$

For a finite, irreducible and homogeneous CTMC, the steady-state probabilities $\{\pi_j\}$ always exist and are independent of the initial state distribution. They are uniquely given by the solution of the equations:

$$-q_{jj}\pi_j + \sum_{k \neq j} q_{kj}\pi_k = 0 \quad \text{subject to} \quad \sum_i \pi_i = 1$$

Again, this can be expressed in matrix vector form (in terms of the vector $\boldsymbol{\pi}$ with elements $\{\pi_1, \pi_2, \ldots, \pi_N\}$ and the matrix $\mathbf{Q}$ defined above) as:

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0} \tag{2}$$

A CTMC also has an embedded discrete-time Markov chain (EMC) which describes the behaviour of the chain at state-transition instants, that is to say the probability that the next state is j given that the current state is i. The EMC of a CTMC has a one-step $N \times N$ transition matrix $\mathbf{P}$ where $p_{ij} = -q_{ij}/q_{ii}$ for $i \neq j$ and $p_{ij} = 0$ for $i = j$.

The steady-state distribution enables us to compute various basic resource-based measures (such as utilisation, mean throughput, and so on); however, more advanced response-time measures (such as quantiles of response time) require a *first passage time* analysis.

Definition 5. *Consider a finite, irreducible CTMC with N states $\{1, 2, \ldots, N\}$ and generator matrix $\mathbf{Q}$. If $\chi(t)$ denotes the states of the CTMC at time t ($t \geq 0$) and $N(t)$ denotes the number of state transitions which have occurred by time t, the* first passage time *from a single source marking i into a non-empty set of target markings $\boldsymbol{j}$ is:*

$$P_{i\boldsymbol{j}}(t) = \inf\{u > 0 : \chi(t+u) \in \boldsymbol{j}, N(t+u) > N(t), \chi(t) = i\}$$

When the CTMC is stationary and time-homogeneous this quantity is independent of t:

$$P_{i\boldsymbol{j}} = \inf\{u > 0 : \chi(u) \in \boldsymbol{j}, N(u) > 0, \chi(0) = i\} \tag{3}$$

That is, the first time the system enters a state in the set of target states $\boldsymbol{j}$, given that the system began in the source state i and at least one state transition has occurred. $P_{i\boldsymbol{j}}$ is a random variable with probability density function $f_{i\boldsymbol{j}}(t)$ such that:

$$\mathbb{P}(t_1 < P_{i\boldsymbol{j}} < t_2) = \int_{t_1}^{t_2} f_{i\boldsymbol{j}}(t)\ \mathrm{d}t \qquad \text{for } 0 \leq t_1 < t_2$$

In order to determine $f_{i\boldsymbol{j}}(t)$ it is necessary to convolve the state holding-time density functions over all possible paths (including cycles) from state i to all of the states in $\boldsymbol{j}$.

The calculation of the convolution of two functions in t-space can be more easily accomplished by multiplying their Laplace transforms together in s-space and inverting the result. The calculation of $f_{i\boldsymbol{j}}(t)$ is therefore achieved by calculating the Laplace transform of the convolution of the state holding times over all paths between i and $\boldsymbol{j}$ and then numerically inverting this Laplace transform (see Sect. 4.3 for a description of two inversion algorithms).

In a CTMC all state sojourn times are exponentially distributed, so the density function of the sojourn time in state i is $\mu_i e^{-\mu_i t}$, where $\mu_i = -q_{ii}$ (as before). The Laplace transform of an exponential density function with rate parameter λ is:

$$L\{\lambda e^{-\lambda t}\} = \frac{\lambda}{\lambda + s}$$

Denoting the Laplace transform of the density function $f_{i\boldsymbol{j}}(t)$ of the passage time random variable $P_{i\boldsymbol{j}}$ as $L_{i\boldsymbol{j}}(s)$, we proceed by means of a first-step analysis. That is, to calculate the first passage time from state i into the set of target states $\boldsymbol{j}$, we consider moving from state i to its set of direct successor states $\boldsymbol{k}$ and thence from states in $\boldsymbol{k}$ to states in $\boldsymbol{j}$. This can be expressed as the following system of linear equations:

$$L_{i\boldsymbol{j}}(s) = \sum_{k \notin \boldsymbol{j}} p_{ik} \left(\frac{-q_{ii}}{s - q_{ii}} \right) L_{k\boldsymbol{j}}(s) + \sum_{k \in \boldsymbol{j}} p_{ik} \left(\frac{-q_{ii}}{s - q_{ii}} \right) \tag{4}$$

The first term (i.e. the summation over non-target states $k \notin \boldsymbol{j}$) convolves the sojourn time density in state i with the density of the time taken for the system to evolve from state k into a target state in $\boldsymbol{j}$, weighted by the probability that the system transits from state i to state k. The second term (i.e. the summation over target states $k \in \boldsymbol{j}$) simply reflects the sojourn time density in state i weighted by the probability that a transition from state i into a target state k occurs.

Given that $p_{ij} = -q_{ij}/q_{ii}$ in the context of a CTMC, Eq. (4) can be rewritten more simply as:

$$L_{i\boldsymbol{j}}(s) = \sum_{k \notin \boldsymbol{j}} \frac{q_{ik}}{s - q_{ii}} L_{k\boldsymbol{j}}(s) + \sum_{k \in \boldsymbol{j}} \frac{q_{ik}}{s - q_{ii}} \tag{5}$$

This set of linear equations can be expressed in matrix–vector form. For example, when $\boldsymbol{j} = \{1\}$ we have:

$$\begin{pmatrix} s - q_{11} & -q_{12} & \cdots & -q_{1n} \\ 0 & s - q_{22} & \cdots & -q_{2n} \\ 0 & -q_{32} & \cdots & -q_{3n} \\ 0 & \vdots & \ddots & \vdots \\ 0 & -q_{n2} & \cdots & s - q_{nn} \end{pmatrix} \begin{pmatrix} L_{1\boldsymbol{j}}(s) \\ L_{2\boldsymbol{j}}(s) \\ L_{3\boldsymbol{j}}(s) \\ \vdots \\ L_{n\boldsymbol{j}}(s) \end{pmatrix} = \begin{pmatrix} 0 \\ q_{21} \\ q_{31} \\ \vdots \\ q_{n1} \end{pmatrix} \tag{6}$$

Our formulation of the passage time quantity in Eq. (3) states that we must observe at least one state-transition during the passage. In the case where $i \in \boldsymbol{j}$ (as for $L_{1\boldsymbol{j}}(s)$ in the above example), we therefore calculate the density of the cycle time to return to state i rather than requiring $L_{i\boldsymbol{j}}(s) = 1$.

Given a particular (complex-valued) s, Eq. (5) can be solved for $L_{i\boldsymbol{j}}(s)$ by standard iterative numerical techniques for the solution of systems of linear equations in $\mathbf{Ax} = \mathbf{b}$ form. Many numerical Laplace transform inversion algorithms (such as the Euler and Laguerre methods) can identify in advance the s-values at which $L_{i\boldsymbol{j}}(s)$ must be calculated in order to perform the numerical inversion. Therefore, if the algorithm requires m different values of $L_{i\boldsymbol{j}}(s)$, Eq. (5) will need to be solved m times.

The corresponding cumulative distribution function $F_{i\boldsymbol{j}}(t)$ of the passage time is obtained by integrating under the density function. This integration can be achieved in terms of the Laplace transform of the density function by dividing it by s, i.e. $F^*_{i\boldsymbol{j}}(s) = L_{i\boldsymbol{j}}(s)/s$. In practice, if Eq. (5) is solved as part of the inversion process for calculating $f_{i\boldsymbol{j}}(t)$, the m values of $L_{i\boldsymbol{j}}(s)$ can be retained. Once the numerical inversion algorithm has used them to compute $f_{i\boldsymbol{j}}(t)$, these values can be recovered, divided by s and then taken as input by the numerical inversion algorithm again to compute $F_{\boldsymbol{i}\boldsymbol{j}}(t)$. Thus, in calculating $f_{\boldsymbol{i}\boldsymbol{j}}(t)$, we get $F_{\boldsymbol{i}\boldsymbol{j}}(t)$ for little further computational effort.

When there are multiple source markings, denoted by the vector $\boldsymbol{i}$, the Laplace transform of the response time density at equilibrium is:

$$L_{\boldsymbol{i}\,\boldsymbol{j}}(s) = \sum_{k \in \boldsymbol{i}} \alpha_k L_{k\boldsymbol{j}}(s)$$

where the weight α_k is the equilibrium probability that the state is $k \in \boldsymbol{i}$ at the starting instant of the passage. This instant is the moment of entry into state k; thus α_k is proportional to the equilibrium probability of the state k in the underlying embedded (discrete-time) Markov chain (EMC) of the CTMC with one-step transition matrix $\mathbf{P}$ as defined in Sect. 2.1. That is:

$$\alpha_k = \begin{cases} \pi_k / \sum_{j \in \boldsymbol{i}} \pi_j & \text{if } k \in \boldsymbol{i} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where the vector $\boldsymbol{\pi}$ is any non-zero solution to $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$. The row vector with components α_k is denoted by $\boldsymbol{\alpha}$.

Uniformisation Passage time densities and quantiles in CTMCs may also be computed through the use of *uniformisation* (also known as *randomisation*) [3–8]. This transforms a CTMC into one in which all states have the same mean holding time $1/q$, by allowing "invisible" transitions from a state to itself. This is equivalent to a discrete-time Markov chain, after normalisation of the rows, together with an associated Poisson process of rate q.

Definition 6. *The one-step transition probability matrix* $\mathbf{P}$ *which characterises the one-step behaviour of a uniformised DTMC is derived from the generator matrix* $\mathbf{Q}$ *of the CTMC as:*

$$\mathbf{P} = \mathbf{Q}/q + \mathbf{I} \tag{8}$$

where the rate $q > \max_i |q_{ii}|$ *ensures that the DTMC is aperiodic by guaranteeing that there is at least one single-step transition from a state to itself.*

We ensure that only the first passage time density is calculated and that we do not consider the case of successive visits to a target state by making the target states in $\mathbf{P}$ absorbing. We denote by $\mathbf{P}'$ the one-step transition probability matrix of the modified, uniformised chain.

The calculation of the first passage time density between two states then has two main components. The first considers the time to complete n hops ($n = 1, 2, 3, \ldots$). Recall that in the uniformised chain all transitions occur with rate q. The density of the time taken to move between two states is found by convolving the state holding-time densities along all possible paths between the states. In a standard CTMC, convolving holding times in this manner is non-trivial as, although they are all exponentially distributed, their rate parameters are different. In a CTMC which has undergone uniformisation, however, all states have exponentially-distributed state holding-times with the same parameter q. This means that the convolution of n of these holding-time densities is an n-stage Erlang density with rate parameter q.

Secondly, it is necessary to calculate the probability that the transition between a source and target state occurs in exactly n hops of the uniformised chain, for every value of n between 1 and a maximum value m. The value of m is determined when the value of the nth Erlang density function (the left-hand term in Eq. (9)) drops below some threshold value. After this point, further terms are deemed to add nothing significant to the passage time density and so are disregarded.

The density of the time to pass between a source state i and a target state j in a uniformised Markov chain can therefore be expressed as the sum of m n-stage Erlang densities, weighted with the probability that the chain moves from state i to state j in exactly n hops ($1 \leq n \leq m$). This can be generalised to allow for multiple target states in a straightforward manner; when there are multiple source states it is necessary to provide a probability distribution across this set of states (such as the renormalised steady-state distribution calculated below in Eq. (11)).

The response time between the non-empty set of source states $\boldsymbol{i}$ and the non-empty set of target states $\boldsymbol{j}$ in the uniformised chain therefore has probability density function:

$$\begin{aligned} f_{\boldsymbol{ij}}(t) &= \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \\ &\simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \qquad (9) \end{aligned}$$

where:

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P}' \qquad \text{for } n \geq 0 \qquad (10)$$

with:

$$\pi_k^{(0)} = \begin{cases} 0 & \text{for } k \notin \boldsymbol{i} \\ \pi_k / \sum_{j \in \boldsymbol{i}} \pi_j & \text{for } k \in \boldsymbol{i} \end{cases} \qquad (11)$$

The π_k values are the steady state probabilities of the corresponding state k in the CTMC's embedded Markov chain. When the convergence criterion:

$$\frac{\|\boldsymbol{\pi}^{(n)} - \boldsymbol{\pi}^{(n-1)}\|_\infty}{\|\boldsymbol{\pi}^{(n)}\|_\infty} < \varepsilon \qquad (12)$$

is met, for given tolerance ε, the vector $\boldsymbol{\pi}^{(n)}$ is considered to have converged and no further multiplications with $\mathbf{P}'$ are performed. Here, $\|\mathbf{x}\|_\infty$ is the infinity-norm given by $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

The corresponding cumulative distribution function for the passage time, $F_{\boldsymbol{ij}}(t)$, can be calculated by substituting the cumulative distribution function for the Erlang distribution into Eq. (9) in place of the Erlang density function term, viz.:

$$\begin{aligned} F_{\boldsymbol{ij}}(t) &= \sum_{n=1}^{\infty} \left(\left(1 - e^{-qt} \sum_{k=0}^{n-1} \frac{(qt)^k}{k!} \right) \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \\ &\simeq \sum_{n=1}^{m} \left(\left(1 - e^{-qt} \sum_{k=0}^{n-1} \frac{(qt)^k}{k!} \right) \sum_{k \in \boldsymbol{j}} \pi_k^{(n)} \right) \end{aligned}$$

where $\boldsymbol{\pi}^{(n)}$ is defined as in Eqs. (10) and (11).

2.2 Semi-Markov Processes

Semi-Markov Processes (SMPs) are an extension of Markov processes which allow for generally distributed sojourn times. Although the memoryless property no longer holds for state sojourn times, at transition instants SMPs still behave in the same way as Markov processes (that is to say, the choice of the next state is based only on the current state) and so share some of their analytical tractability.

Definition 7. *Consider a Markov renewal process $\{(\chi_n, T_n) : n \geq 0\}$ where T_n is the time of the nth transition ($T_0 = 0$) and $\chi_n \in \mathcal{S}$ is the state at the nth transition. Let the kernel of this process be:*

$$R(n, i, j, t) = \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i)$$

for $i, j \in \mathcal{S}$. The continuous time semi-Markov process, $\{Z(t), t \geq 0\}$, defined by the kernel R, is related to the Markov renewal process by:

$$Z(t) = \chi_{N(t)}$$

where $N(t) = \max\{n : T_n \leq t\}$, i.e. the number of state transitions that have taken place by time t. Thus $Z(t)$ represents the state of the system at time t.

We consider only time-homogeneous SMPs in which $R(n, i, j, t)$ is independent of n, that is for:

$$\begin{aligned} R(i, j, t) &= \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i) \quad \text{for any } n \geq 0 \\ &= p_{ij} H_{ij}(t) \end{aligned}$$

where $p_{ij} = \mathbb{P}(\mathcal{X}_{n+1} = j \mid \mathcal{X}_n = i)$ is the state transition probability between states i and j and $H_{ij}(t) = \mathbb{P}(T_{n+1} - T_n \leq t \mid \mathcal{X}_{n+1} = j, \mathcal{X}_n = i)$, is the sojourn time distribution in state i when the next state is j. An SMP can therefore be characterised by two matrices $\mathbf{P}$ and $\mathbf{H}$ with elements p_{ij} and H_{ij} respectively.

Semi-Markov processes can be analysed for steady-state performance metrics in a similar manner as DTMCs and CTMCs. To do this, we need to know the steady-state probabilities of the SMP's embedded Markov chain and the average time spent in each state. The first of these can be calculated by solving $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$, as in the case of DTMCs. The average time in state i, $\mathbb{E}[\tau_i]$, is the weighted sum of the averages of the sojourn time in the state i when going to state j, $\mathbb{E}[\tau_{ij}]$, for all successor states j of i, that is:

$$\mathbb{E}[\tau_i] = \sum_j p_{ij}\mathbb{E}[\tau_{ij}]$$

The steady-state probability of being in state i of the SMP is then:

$$\phi_i = \frac{\pi_i \mathbb{E}[\tau_i]}{\sum_{m=1}^{N} \pi_m \mathbb{E}[\tau_m]} \tag{13}$$

That is, the long-run probability of finding the SMP in state i is the probability of its EMC being in state i multiplied by the average amount of time the SMP spends in state i, normalised over the mean total time spent in all of the states of the SMP.

Passage-time analysis for SMPs is also possible by extending the Laplace transform method for CTMCs to cater for generally-distributed state sojourn times.

Definition 8. *Consider a finite, irreducible, continuous-time semi-Markov process with N states $\{1, 2, \ldots, N\}$. Recalling that $Z(t)$ denotes the state of the SMP at time t $(t \geq 0)$, the first passage time from a source state i at time t into a non-empty set of target states $\boldsymbol{j}$ is:*

$$P_{i\boldsymbol{j}}(t) = \inf\{u > 0 : Z(t+u) \in \boldsymbol{j}, N(t+u) > N(t) \mid Z(t) = i\} \tag{14}$$

For a stationary time-homogeneous SMP, $P_{i\boldsymbol{j}}(t)$ is independent of t and we have:

$$P_{i\boldsymbol{j}} = \inf\{u > 0 : Z(u) \in \boldsymbol{j}, N(u) > 0 \mid Z(0) = i\} \tag{15}$$

$P_{i\boldsymbol{j}}$ has an associated probability density function $f_{i\boldsymbol{j}}(t)$ such that the passage time quantile is given as:

$$\mathbb{P}(t_1 < P_{i\boldsymbol{j}} < t_2) = \int_{t_1}^{t_2} f_{i\boldsymbol{j}}(t)\,\mathrm{d}t \qquad \text{for } 0 \leq t_1 < t_2 \tag{16}$$

In general, the Laplace transform of $f_{i\boldsymbol{j}}$, $L_{i\boldsymbol{j}}(s)$, can be computed by solving a set of N linear equations:

$$L_{i\boldsymbol{j}}(s) = \sum_{k \notin \boldsymbol{j}} r^*_{ik}(s) L_{k\boldsymbol{j}}(s) + \sum_{k \in \boldsymbol{j}} r^*_{ik}(s) \qquad \text{for } 1 \leq i \leq N \tag{17}$$

where $r^*_{ik}(s)$ is the Laplace-Stieltjes transform (LST) of $R(i,k,t)$ and is defined by:

$$r^*_{ik}(s) = \int_0^\infty e^{-st}\,\mathrm{d}R(i,k,t) \tag{18}$$

Eq. (17) has a matrix–vector form where the elements of the matrix are arbitrary complex functions; care needs to be taken when storing such functions for eventual numerical inversion (see Sect. 4.3). For example, when $\boldsymbol{j} = \{1\}$, Eq. (17) yields:

$$\begin{pmatrix} 1 & -r^*_{12}(s) & \cdots & -r^*_{1N}(s) \\ 0 & 1-r^*_{22}(s) & \cdots & -r^*_{2N}(s) \\ 0 & -r^*_{32}(s) & \cdots & -r^*_{3N}(s) \\ \vdots & \vdots & \ddots & \vdots \\ 0 & -r^*_{N2}(s) & \cdots & 1-r^*_{NN}(s) \end{pmatrix} \begin{pmatrix} L_{1\boldsymbol{j}}(s) \\ L_{2\boldsymbol{j}}(s) \\ L_{3\boldsymbol{j}}(s) \\ \vdots \\ L_{N\boldsymbol{j}}(s) \end{pmatrix} = \begin{pmatrix} r^*_{11}(s) \\ r^*_{21}(s) \\ r^*_{31}(s) \\ \vdots \\ r^*_{N1}(s) \end{pmatrix} \tag{19}$$

When there are multiple source states, denoted by the vector $\boldsymbol{i}$, the Laplace transform of the passage time density at steady-state is:

$$L_{\boldsymbol{i}\boldsymbol{j}}(s) = \sum_{k \in \boldsymbol{i}} \alpha_k L_{k\boldsymbol{j}}(s) \tag{20}$$

where the weight α_k is the probability at equilibrium that the system is in state $k \in \boldsymbol{i}$ at the starting instant of the passage. As with CTMCs $\boldsymbol{\alpha}$ is defined in terms of $\boldsymbol{\pi}$, the steady-state vector of the embedded discrete-time Markov chain with one-step transition probability matrix $\mathbf{P}$:

$$\alpha_k = \begin{cases} \pi_k / \sum_{j \in \boldsymbol{i}} \pi_j & \text{if } k \in \boldsymbol{i} \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

3 Modelling Formalisms

Stochastic models are specified using graphical or symbolic languages known as *modelling formalisms*. Below we describe two popular formalisms: Stochastic Petri nets and Stochastic Process Algebras.

3.1 Stochastic Petri Nets

We briefly outline two types of stochastic Petri net: Generalised Stochastic Petri Nets (GSPNs) which allow timed exponential and immediate transitions, and Semi-Markov Stochastic Petri Nets (SM-SPNs) which specify models with generally distributed transitions.

Generalised Stochastic Petri Nets Generalised Stochastic Petri nets are an extension of Place-Transition nets, which are ordinary, untimed Petri nets. A Place-Transition net does not have firing delays associated with its transitions and is formally defined in [2]:

Definition 9. *A* Place-Transition net *is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$ *where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places.*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions.*
- $P \cap T = \emptyset$.
- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are the backward and forward incidence functions, respectively. If* $I^-(p, t) > 0$*, an arc leads from place* p *to transition* t*, and if* $I^+(p, t) > 0$ *then an arc leads from transition* t *to place* p.
- $M_0 : P \rightarrow \mathbb{N}_0$ *is the initial marking defining the initial number of tokens on every place.*

A marking is a vector of integers representing the number of tokens on each place in a Petri net. The set of all markings that are reachable from the initial marking M_0 is known as the *state space* or *reachability set* of the Petri net, and is denoted by $R(M_0)$. The connections between markings in the reachability set form the *reachability graph.* Formally, if the firing of a transition that is enabled in marking M_i results in marking M_j, then the reachability graph contains a directed arc from marking M_i to marking M_j.

GSPNs [9] are timed extensions of Place-Transition nets with two types of transitions: *immediate* transitions and *timed* transitions. Once enabled, immediate transitions fire in zero time, while timed transitions fire after an exponentially distributed firing delay. Firing of immediate transitions has priority over the firing of timed transitions.

The formal definition of a GSPN is as follows [2]:

Definition 10. *A GSPN is a 4-tuple* $GSPN = (PN, T_1, T_2, W)$ *where*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
- $T_2 \subset T$ *denotes the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$*,* $T = T_1 \cup T_2$
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* w_i *is either*
  - *a (possibly marking dependent)* **rate** $\in \mathbb{R}^+$ *of an exponential distribution specifying the firing delay, when transition* t_i *is a timed transition, i.e.* $t_i \in T_1$
    or
  - *a (possibly marking dependent)* **weight** $\in \mathbb{R}^+$ *specifying the relative firing frequency, when transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$.

The reachability graph of a GSPN contains two types of markings. A vanishing marking is one in which an immediate transition is enabled. The sojourn time in such markings is zero. A tangible marking is one which enables only timed transitions. The sojourn time in such markings is exponentially distributed. Once vanishing markings have been eliminated (see [10] for a discussion of methods for vanishing state elimination), the resulting tangible reachability graph of a GSPN maps directly onto a CTMC.

Semi-Markov Stochastic Petri Nets Semi-Markov stochastic Petri nets [11] (SM-SPNs) are extensions of GSPNs which support arbitrary holding-time distributions and which generate an underlying semi-Markov process rather than a Markov process. Note that it is not intended that they be a novel technique for dealing with concurrently-enabled generally-distributed transitions. They are instead a useful high-level vehicle for the construction of large semi-Markov models for analysis.

Definition 11. *An SM-SPN consists of a 4-tuple,* $(PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$*, where:*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.* P *is the set of places,* T*, the set of transitions,* $I^{+/-}$ *are the forward and backward incidence functions describing the connections between places and transitions and* M_0 *is the initial marking.*
- $\mathcal{P} : T \times \mathcal{M} \rightarrow \mathbb{Z}^+$*, denoted* $p_t(m)$*, is a marking-dependent priority function for a transition.*
- $\mathcal{W} : T \times \mathcal{M} \rightarrow \mathbb{R}^+$*, denoted* $w_t(m)$*, is a marking-dependent weight function for a transition, to allow implementation of probabilistic choice.*
- $\mathcal{D} : T \times \mathcal{M} \rightarrow (\mathbb{R}^+ \rightarrow [0, 1])$*, denoted* $d_t(m)$*, is a marking-dependent cumulative distribution function for the firing time of a transition.*

In the above, $\mathcal{M}$ *is the set of all markings for a given net. Further, we define the following general net-enabling functions:*

- $\mathcal{E}_N : \mathcal{M} \rightarrow P(T)$*, a function that specifies net-enabled transitions from a given marking.*
- $\mathcal{E}_P : \mathcal{M} \rightarrow P(T)$*, a function that specifies priority-enabled transitions from a given marking.*

The net-enabling function, $\mathcal{E}_N$, is defined in the usual way for standard Petri nets: if all preceding places have occupying tokens then a transition is net-enabled. Similarly, we define the more stringent priority-enabling function, $\mathcal{E}_P$. For a given marking, m, $\mathcal{E}_P(m)$ selects only those net-enabled transitions that have the highest priority, that is:

$$\mathcal{E}_P(m) = \{t \in \mathcal{E}_N(m) : p_t(m) = \max\{p_{t'}(m) : t' \in \mathcal{E}_N(m)\}\} \tag{22}$$

Now for a given priority-enabled transition, $t \in \mathcal{E}_P(m)$, the probability that it will be the one that actually fires after a delay sampled from its firing distribution, $d_t(m)$, is:

$$\mathbb{P}(t \in \mathcal{E}_P(m) \text{ fires}) = \frac{w_t(m)}{\sum_{t' \in \mathcal{E}_P(m)} w_{t'}(m)} \tag{23}$$

Note that the choice of which priority-enabled transition is fired in any given marking is made by a probabilistic selection based on transition weights, and is not a race condition based on finding the minimum of samples extracted from firing-time distributions. This mechanism enables the underlying reachability graph of an SM-SPN to be mapped directly onto a semi-Markov chain.

3.2 Stochastic Process Algebras

A process algebra is an abstract language which differs from the formalisms we have considered so far because it is not based on a notion of *flow*. Instead, systems are modelled as a collection of cooperating *agents* or *processes* which execute atomic *actions*. These actions can be carried out independently or can be synchronised with the actions of other agents.

Since models are typically built up from smaller components using a small set of combinators, process algebras are particularly suited to the modelling of large systems with hierarchical structure. This support for *compositionality* is complemented by mechanisms to provide abstraction and compositional reasoning.

Two of the best known process algebras are Hoare's Communicating Sequential Processes (CSP) [12] and Milner's Calculus of Communicating Systems (CCS) [13]. These algebras do not include a notion of time so they can only be used to determine qualitative correctness properties of systems such as the freedom from deadlock and livelock. Stochastic Process Algebras (SPAs) associate a random variable, representing a time duration, with each action. This addition allows quantitative performance analysis to be carried out on SPA models in the same fashion as for SPNs.

Here we will briefly describe the Markovian SPA, PEPA [14]. Other SPAs include TIPP [15, 16], MPA [17] and EMPA [18] which are similar to PEPA. A detailed comparison of Markovian stochastic process algebras can be found in [19]. More recently developed non-Markovian SPAs allow for generally-distributed delays as part of the model; examples of these include SPADES [20, 21], semi-Markov PEPA [22] and iGSMPA [23, 24].

PEPA models are built from components which perform activities of form (α, r) where α is the action type and $r \in \mathbb{R}^+ \cup \{\top\}$ is the exponentially distributed rate of the action. The special symbol $\top$ denotes an passive activity that may only take place in synchrony with another action whose rate is specified.

Interaction between components is expressed using a small set of combinators, which are briefly described below:

Action prefix: Given a process P, $(\alpha, r).P$ represents a process that performs an activity of type α, which has a duration exponentially distributed with mean $1/r$, and then evolves into P.

Constant definition: Given a process Q, $P \overset{def}{=} Q$ means that P is a process which behaves in exactly the same way as Q.

Competitive choice: Given processes P and Q, $P + Q$ represents a process that behaves either as P or as Q. The current activities of both P and Q are enabled and a race condition determines into which component the process will evolve.

Cooperation: Given processes P and Q and a set of action types L, $P \underset{L}{\bowtie} Q$ defines the concurrent synchronised execution of P and Q over the cooperation set L. No synchronisation takes place for any activity $\alpha \notin L$, so such activities can take place independently. However, an activity $\alpha \in L$ only occurs when both P and Q are capable of performing the action. The rate at which the action occurs is given by the minimum of the rates at which the two components would have executed the action in isolation.
Cooperation over the empty set $P \underset{\emptyset}{\bowtie} Q$ represents the independent concurrent execution of processes P and Q and is denoted by $P \,||\, Q$.

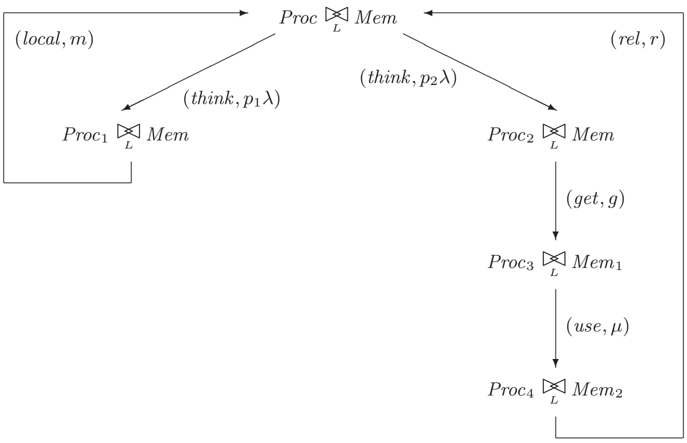
Encapsulation: Given a process P and a set of actions L, P/L represents a process that behaves like P except that activities $\alpha \in L$ are hidden and performed as a *silent* activity. Such activities cannot be part of a cooperation set.

PEPA specifications can be mapped onto continuous time Markov chains in a straightforward manner. Based on the labelled transition system semantics that are normally specified for a process algebra system, a transition diagram or *derivation graph* can be associated with any language expression. This graph describes all possible evolutions of a system and, like a tangible reachability graph in the context of GSPNs, is isomorphic to a CTMC which can be solved for its steady-state distribution. Fig. 1 shows a PEPA specification of a multiprocessor system together with its corresponding derivation graph.

4 Methods for Tackling Large Unstructured State Spaces

We proceed to review several approaches to the problem of analysing stochastic models with large underlying state spaces, covering the major phases in an advanced performance analysis pipeline, i.e. state generation, steady-state solution and passage-time analysis.

The methods reviewed here are based on explicit state representation, and so are particularly suited to the analysis of systems with large unstructured state



$Proc \underset{L}{\bowtie} Mem$
$(local, m)$
(rel, r)
$(think, p_1\lambda)$
$(think, p_2\lambda)$
$Proc_1 \underset{L}{\bowtie} Mem$
$Proc_2 \underset{L}{\bowtie} Mem$
(get, g)
$Proc_3 \underset{L}{\bowtie} Mem_1$
(use, μ)
$Proc_4 \underset{L}{\bowtie} Mem_2$

$$\begin{aligned}
&Proc \overset{def}{=} (think, p_1\lambda).Proc_1 + (think, p_2\lambda).Proc_2\\
&Proc_1 \overset{def}{=} (local, m).Proc\\
&Proc_2 \overset{def}{=} (get, g).Proc_3 \quad Proc_3 \overset{def}{=} (use, \mu).Proc_4 \quad Proc_4 \overset{def}{=} (rel, r).Proc\\
&Mem \overset{def}{=} (get, \top).Mem_1 \quad Mem_1 \overset{def}{=} (use, \mu).Mem_2 \quad Mem_2 \overset{def}{=} (rel, \top).Mem\\
&Sys_4 \overset{def}{=} Proc \underset{L}{\bowtie} Mem \qquad \text{where } L = \{get, rel, use\}
\end{aligned}$$

Fig. 1. A PEPA specification and its corresponding derivation graph [25]

spaces. We note that there are effective approaches based on implicit/symbolic state representation which can be applied to systems whose underlying state spaces are structured in some way – for example methods based on Binary Decision Diagrams and related data structures [26, 27, 7, 28], and Kronecker methods [29]. Since these methods are the subjects of other chapters in this volume, they are not discussed further here.

4.1 Probabilistic State Space Generation

The first challenge in the quantitative analysis of stochastic models is to generate all reachable states or configurations that the system can enter. The main obstacle to this task is the huge number of states that can emerge, a problem compounded by the large size of individual state descriptors. Consequently there are severe memory and time constraints on the number of states that can be generated using a simplistic explicit exhaustive enumeration.

The Case for Probabilistic Algorithms A useful, but at first seemingly bizarre, method of dealing with a problem that seems to be infeasible (either in terms of computational or storage demands) is to relax the requirement that a solution should always produce the correct answer. Adopting such a *probabilistic* or *randomised* approach can lead to dramatic memory and time savings. Of course, in order to be useful in practice, the risk of producing an incorrect result must be quantified and kept very small.

One of the most exciting early applications of probabilistic algorithms was in finding an efficient solution to the primality problem (i.e. to determine if some positive integer n is prime). This problem has direct application to public key cryptographic systems, many of which are based on finding a modulus of form pq where p and q are large prime numbers.

The Miller-Rabin primality test [30] provides an efficient probabilistic solution to the primality problem by relying on three facts:

- If n is composite (i.e. not prime) then at least three quarters of the natural numbers less than n are *witnesses* to the compositeness of n (i.e. can be used to establish that n is not prime).
- If n is prime then there is no natural number less than n that is witness to the compositeness of n.
- Given number natural numbers m and n with $m < n$, there is an efficient algorithm which ascertains whether or not m is a witness to the compositeness of n.

The algorithm works by performing k witness tests using randomly chosen natural numbers less than n; should all of these witness tests fail, we assume n is prime. Indeed, if n is prime, this is the correct conclusion. If n is composite, the chances of failing to find a witness (and hence detect that the number is not prime) is 2^{-2k}. Hence, by increasing k, we can arbitrarily increase the reliability of the algorithm at logarithmic run time cost; when k is around 20, the algorithm is probably more reliable than most computer hardware.

Application to State Space Generation While research into probabilistic algorithms for solving the primality problem has been focused on reducing run time, the application of probabilistic algorithms to state space generation has been focused on the need to reduce memory requirements. In particular, the memory consumption of explicit state space generation algorithms is heavily dependent on the layout and management of a data structure known as the *explored state table*. This table prevents redundant work by identifying which states have already been encountered. Its implementation is particularly challenging because the table is accessed randomly and must be able to rapidly store and retrieve information about every reachable state. One approach is to store the full state descriptor of each state in the table. This *exhaustive* approach guarantees full coverage, but at very high memory cost. *Probabilistic* methods use one-way hashing techniques to drastically reduce the amount of memory required to store states. However, this introduces the risk that two distinct states will have the same hashed representation, resulting in the misidentification and omission of states in the state graph. Naturally, it is important to quantify this risk and to find ways of reducing it to an acceptable level.

The next sections review three of the best-known probabilistic methods (interested readers might also like to consult [31] which presents another recent survey). In each case, we include an analysis and discussion of memory consumption and the omission probability.

Holzmann's Bit-state Hashing Holzmann's bit-state hashing (or supertrace) technique [32, 33] was developed in an attempt to maximize state coverage in the face of limited memory. The technique has proved popular because of its elegance and simplicity and has consequently been included in many research and commercial verification tools.

Holzmann's method is based on the use of Bloom Filters. These were conceived by Burton H. Bloom in 1970 as space-efficient probabilistic data structures for testing set membership [34]. Here the explored state table takes the form of a bit vector T. Initially all bits in T are set to zero. States are mapped into positions in this bit vector using a hash function h, so that when state s is inserted into the table its corresponding bit $T[h(s)]$ is set to one. To check whether a state s is already in the table, the value of $T[h(s)]$ is examined. If it is zero, we know that the state has definitely not been previously encountered; otherwise it is assumed that the state has already been explored. This may be a mistake, however, since two distinct states can be hashed onto the same position in the bit vector. The result of a hash collision will be that one of the states will be incorrectly classified as explored, resulting in the omission of one or more states from the state space. Assuming a good hash function which distributes states randomly, the probability of no hash collisions p when inserting n states into a bit vector of t bits is:

$$p = \frac{t!}{(t-n)!t^n} = \prod_{i=0}^{n-1} \frac{(t-i)}{t} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{t}\right)$$

Assuming the favourable case $n \ll t$ and using the approximation $e^x \approx (1+x)$ for $|x| \ll 1$, we obtain:

$$p \approx \prod_{i=0}^{n-1} e^{-i/t} = e^{\sum_{i=0}^{n-1} -i/t} = e^{-\frac{n(n-1)}{2t}} = e^{\frac{n-n^2}{2t}}$$

Since $n^2 \gg n$ for large n, a good approximation for p is given by:

$$p \approx e^{-\frac{n^2}{2t}}$$

The corresponding probability of state omission is $q = 1 - p$. Unfortunately the table sizes required to keep the probability of state omission very low are impractically large. For example, to obtain a state omission probability of 0.1% when inserting $n = 10^6$ states requires the allocation of a bit vector of 125TB. The situation can be improved a little by using two independent hash functions h_1 and h_2. When inserting a state s, both $T[h_1(s)]$ and $T[h_2(s)]$ are set to one. Likewise, we conclude s has been explored only if both $T[h_1(s)]$ and $T[h_2(s)]$ are set to one. Wolper and Leroy [35] show that now the probability of no hash collisions is:

$$p \approx e^{-\frac{4n^3}{t^2}}.$$

However the table sizes required to keep the probability of state omission low are still impractically large. Using more than two hash functions helps improve the probability slightly; in fact it turns out that the optimal number of functions is about 20 [35]. However, computing 20 independent hash functions on every state is expensive and the resulting algorithm is very slow. The strength of Holzmann's algorithm therefore lies in the goal for which it was originally designed, i.e. the ability to maximize coverage in the face of limited memory, and not in its ability to provide complete state coverage.

Wolper and Leroy's Hash Compaction Holzmann's method requires a very low ratio of states to hash table entries to provide a good probability of complete state space coverage. Consequently, a large amount of the space allocated to the bit vector will be wasted. Wolper and Leroy observed that it would be better to store which bit positions in the table are occupied instead [35]. This can be done by hashing states onto compressed keys of b bits. These keys can then be stored in a smaller hash table which supports a collision resolution scheme.

Given a hash table with $m \geq n$ slots, the memory required is:

$$M = (mb + m)/8 = m(b + 1)/8$$

since we need to store the keys, as well as a bit vector indicating which hash table slots are occupied. If we wish to construct the state graph efficiently, states also need to be assigned unique state sequence numbers. Given s-bit state sequence numbers, total memory consumption in this case is:

$$M = m(b + s + 1)/8.$$

In terms of the reliability of the technique, this approach is equivalent to a bit-state hashing scheme with a table size of 2^b, so the probability of no collision p is given by:

$$p \approx e^{-\frac{n^2}{2^{b+1}}}$$

Wolper and Leroy recommend compressed values of $b = 64$ bits, i.e. 8-byte compression.

Stern and Dill's Improved Hash Compaction Wolper and Leroy do not discuss exactly how states are mapped onto slots in their hash table. It seems to be implicitly assumed that the hash values used to determine where to store the b-bit compressed values in the hash table are calculated using the b-bit compressed values themselves. Stern and Dill [36] noticed that the omission probability can be dramatically reduced in two ways – firstly by calculating the hash values and compressed values independently and secondly by using a collision resolution scheme which keeps the number of probes per insertion low. This improved technique is so effective that it requires only 5 bytes per state in situations where Wolper and Leroy's standard hash compaction requires 8 bytes per state.

Given a hash table with m slots, states are inserted into the table using two hash functions $h_1(s)$ and $h_2(s)$. These hash functions generate the probe sequence $h^{(0)}(s), h^{(1)}(s), \ldots, h^{(m-1)}(s)$ with $h^{(i)}(s) = (h_1(s) + ih_2(s)) \bmod m$ for $i = 0, 1, \ldots, m-1$. This double hashing scheme prevents the clustering associated with simple rehashing algorithms such as linear probing. A separate independent compression function h_3 is used to calculate the b-bit compressed state values which are stored in the table.

Slots are examined in the order of the probe sequence, until one of two conditions are met:

1. If the slot currently being examined is empty, the compressed value is inserted into the table at that slot.
2. If the slot is occupied by a compressed value equal to the $h_3(s)$, we assume (possibly incorrectly) that the state has already been explored.

Total memory consumption is the same as for Wolper and Leroy's hash compaction method, i.e.

$$M = m(b + s + 1)/8$$

where we assume a bit vector indicates which hash slots are used, and s-bit unique state sequence numbers are used to identify states for efficient construction of the state graph.

Given m slots in the hash table, n of which are occupied by states, Stern and Dill prove that the probability of no state omissions p is given by

$$p \approx \prod_{k=0}^{n-1} \left[\sum_{j=0}^{k} \left(\frac{2^b - 1}{2^b} \right)^j \frac{m-k}{m-j} \prod_{i=0}^{j-1} \frac{k-i}{m-i} \right]$$

This formula takes $O(n^3)$ operations to evaluate. Stern and Dill derive an $O(1)$ approximation given by

$$p \approx \left(\frac{2^b - 1}{2^b} \right)^{(m+1)\ln(\frac{m+1}{m-n+1}) - \frac{n}{2(m-n+1)} + \frac{2n+2mn-n^2}{12(m+1)(m-n+1)^2} - n}$$

An upper bound for the probability of state omission q is

$$q \leq \frac{1}{2^b}\left[(m+1)(H_{m+1} - H_{m-n+1}) - n\right]$$

where $H_n = \sum_{k=1}^{n} 1/k$ is the nth harmonic number [36]. This probability rises sharply as the hash table becomes full, since compressed states being inserted are compared against many compressed values before an empty slot is found. Stern and Dill derive a more straightforward formula for the approximate maximum omission probability for a full table (i.e. with $m = n$):

$$q \approx \frac{1}{2^b} m(\ln m - 1)$$

which shows the omission probability is approximately proportional to $m \ln m$. Increasing b, the number of bits per state, by one roughly halves the maximum omission probability.

Dynamic Probabilistic State Space Generation We now discuss a probabilistic technique which uses dynamic storage allocation and which yields a very low collision probability [37]. The system is illustrated in Fig. 2. Here, the explored state table takes the form of a hash table with several rows. Attached to each row is a linked list which stores compressed state descriptors and state sequence numbers.

Two independent hash functions are used. Given a state descriptor s, the *primary* hash function $h_1(s)$ is used to determine which hash table row should be used to store a compressed state, while the *secondary* hash function $h_2(s)$ is used to compute a compressed state descriptor value (also known as a secondary key). If a state's secondary key $h_2(s)$ is present in the hash table row given by its primary key $h_1(s)$, then the state is deemed to be the already-explored state identified by the sequence number $\mathrm{id}(s)$. Otherwise, the secondary key and a new sequence number are added to the hash table row and the state's successors are added onto the FIFO queue.

Fig. 3 shows the complete sequential dynamic probabilistic state space generation algorithm based on our hash compaction technique. Here H represents the state hash table in which each state $s \in E$ has an entry of form $[h_1(s), h_2(s)]$. Since it is now not necessary to store the full state space E in memory, the insertion of states into E can be handled by writing the states to a disk file as they are encountered.

Note that two states s_1 and s_2 are classified as being equal if and only if $h_1(s_1) = h_1(s_2)$ and $h_2(s_1) = h_2(s_2)$. This may happen even when the two states are different, so collisions may occur (as in all other probabilistic methods). However, as we will see below, the probability of such a collision can be kept very small – certainly much smaller than the chance of a serious man-made error in the specification of the model. In addition, by regenerating the state space with

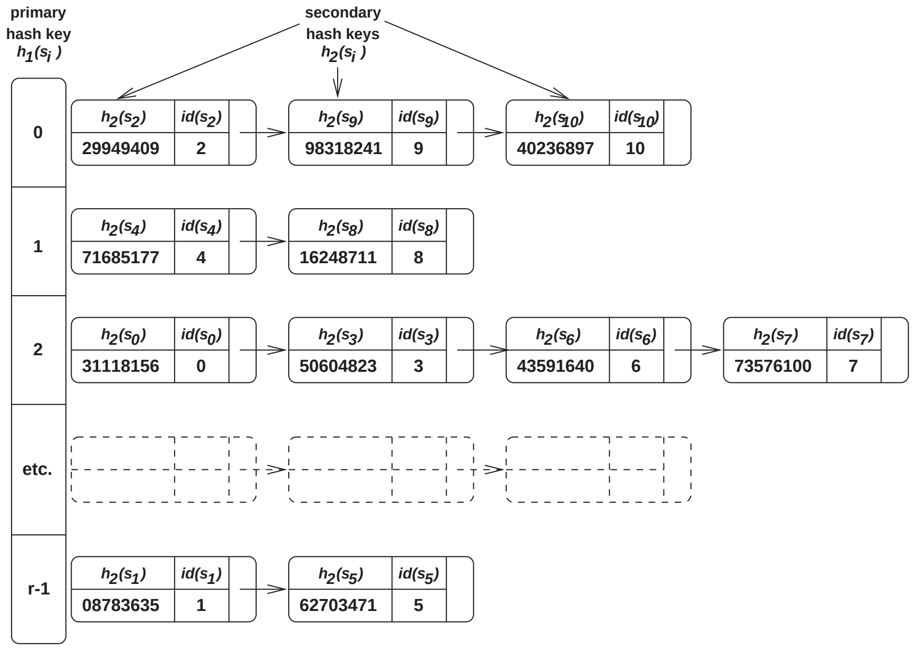


Fig. 2. Layout of the explored state table under the dynamic probabilistic hash compaction scheme

```
begin
    $H = \{[h_1(s_0), h_2(s_0)]\}$
    $F$.add($s_0$)
    $E = \{\ s_0\ \}$
    $A = \emptyset$
    while ($F$ not empty) do begin
        $F$.remove($s$)
        for each $s' \in \text{succ}(s)$ do begin
            if $[h_1(s'), h_2(s')] \notin H$ do begin
                $F$.add($s'$)
                $E = E \cup \{s'\}$
                $H = H \cup \{[h_1(s'), h_2(s')]\}$
            end
            $A = A \cup \{\text{id}(s) \rightarrow \text{id}(s')\}$
        end
    end
end
```

Fig. 3. Sequential dynamic probabilistic state space generation algorithm

different sets of independent hash functions and comparing the resulting number of states and transitions, it is possible to further arbitrarily decrease the risk of an undetected collision.

We now calculate the probability of complete state coverage p. We consider a hash table with r rows and $t = 2^b$ possible secondary key values, where b is the number of bits used to store the secondary key. In such a hash table, there are rt possible ways of representing a state. Assuming that $h_1(s)$ and $h_2(s)$ distribute states randomly and independently, each of these representations are equally likely. Thus, if there are n distinct states to be inserted into the hash table, the probability p that all states are uniquely represented is given by:

$$p = \frac{(rt)!}{(rt-n)!(rt)^n} \tag{24}$$

An equivalent formulation of Eq. (24) is:

$$p = \prod_{i=0}^{n-1} \frac{rt-i}{rt} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{rt}\right) \tag{25}$$

Assuming $n \ll rt$ and using the fact that $e^x \approx (1+x)$ for $|x| \ll 1$, we obtain:

$$p \approx \prod_{i=0}^{n-1} e^{-i/rt} = e^{\sum_{i=0}^{n-1} -i/rt} = e^{-\frac{n(n-1)}{2rt}} = e^{\frac{n-n^2}{2rt}}$$

Since $n^2 \gg n$ for large n, a simple approximation for p is given by:

$$p \approx e^{-\frac{n^2}{2rt}} \tag{26}$$

It can be shown that if $n^2 \ll rt$ then this approximation is also a lower bound for p (and thus provides a conservative estimate for the probability of complete state coverage) [10].

The corresponding upper bound for the probability q that all states are not uniquely represented, resulting in the omission of one or more states from the state space, is of course simply:

$$q = 1 - p \leq \frac{n^2}{2rt} = \frac{n^2}{r2^{b+1}}. \tag{27}$$

Thus the probability of state omission q is proportional to n^2 and is inversely proportional to the hash table size r. Increasing the size of the compressed state descriptors b by one bit halves the omission probability.

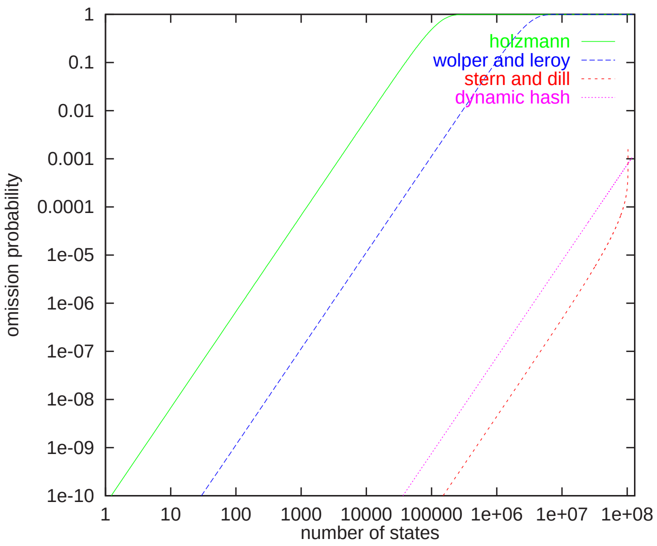


Fig. 4. Contemporary static probabilistic methods compared with the dynamic hash compaction method in terms of omission probability

Method	Parameters
Holzmann	$l = 7.488 \times 10^9$ bits
	$M = 91.4$ MB

Method	Parameters
Wolper and Leroy	$b = 42$ bits
	$s = 32$ bits
	$m = 10^8$ slots
	$M = 91.6$ MB

Method	Parameters
Stern and Dill	$b = 40$ bits
	$s = 32$ bits
	$m = 10.26 \times 10^8$ slots
	$M = 91.4$ MB

Method	Parameters
Dynamic hash	$b = 40$ bits
	$s = 32$ bits
	$r = 6\,000\,000$ rows
	$h = 6$ bytes
	$M = 91.4$ MB (for $n = 10^8$)

Table 1. Parameters used in the comparison of omission probabilities

Comparison of State Omission Probabilities Fig. 4 compares the omission probability of contemporary static probabilistic methods with that of the dynamic hash compaction method for state space sizes of various magnitudes up to 10^8. The parameters used for each method are presented in Tab. 1, and are selected such that the memory use of all four algorithms is the same. The graph shows that the dynamic method yields a far lower omission probability than both Holzmann's method and Wolper and Leroy's method. In addition, the dynamic method is competitive with Stern and Dill's algorithm and yields a better omission probability when the hash table becomes full or nearly full.

Parallel Dynamic Probabilistic State Space Generation We now investigate how our technique can be enhanced to take advantage of the memory and processing power provided by a network of workstations or a distributed-memory parallel computer. We assume there are N nodes available and that each processor has its own local memory and can communicate with other nodes via a network.

In the parallel algorithm, the state space is partitioned between the nodes so that each node is responsible for exploring a portion of the state space and for constructing part of the state graph. A partitioning hash function $h_0(s) \rightarrow (0, \ldots, N-1)$ is used to assign states to nodes, such that node i is responsible for exploring the set of states E_i and for constructing the portion of the state graph A_i where:

$$\begin{aligned} E_i &= \{s : h_0(s) = i\} \\ A_i &= \{(s_1 \rightarrow s_2) : h_0(s_1) = i\} \end{aligned}$$

It is important that $h_0(s)$ achieves a good spread of states across nodes in order to achieve good load balance. Naturally, the values produced by $h_0(s)$ should also be independent of those produced by $h_1(s)$ and $h_2(s)$ to enhance the reliability of the algorithm. Guidelines for choosing hash functions which meet these goals are discussed in [10].

The operation of node i in the parallel algorithm is shown in Fig. 5. Each node i has a local FIFO queue F_i used to hold unexplored local states and a hash table H_i representing a compressed version of the set E_i, i.e. those states which have been explored locally. State s is assigned to processor $h_0(s)$, which stores the state's compressed state descriptor $h_2(s)$ in the local hash table row given by $h_1(s)$. As before, it is not necessary to store the complete state space E_i in memory, since states can be written out to a disk file as they are encountered.

Node i proceeds by removing a state from the local FIFO queue and determining the set of successor states. Successor states for which $h_0(s) = i$ are dealt with locally, while other successor states are sent to the relevant remote processors via calls to send-state(k, g, s). Here k is the remote node, g is the identity of the parent state and s is the state descriptor of the child state. The remote processors

```
begin
    if $h_0(s_0) = i$ do begin
        $H_i = \{[h_1(s_0), h_2(s_0)]\}$
        $F_i$.add($s_0$)
        $E_i = \{s_0\}$
    end else
        $H_i = E_i = \emptyset$
    $A_i = \emptyset$
    while (shutdown signal not received) do begin
        if ($F_i$ not empty) do begin
            $s = F_i$.remove()
            for each $s' \in \mathrm{succ}(s)$ do begin
                if $h_0(s') = i$ do begin
                    if $[h_1(s'), h_2(s')] \notin H_i$ do begin
                        $H_i = H_i \cup \{[h_1(s'), h_2(s')]\}$
                        $F_i$.add($s'$)
                        $E_i = E_i \cup \{s'\}$
                    end
                    $A_i = A_i \cup \{\mathrm{id}(s) \rightarrow \mathrm{id}(s')\}$
                end else
                    send-state($h_0(s')$, id($s$), $s'$)
            end
        end
        while (receive-id($g$, $h$)) do
            $A_i = A_i \cup \{g \rightarrow h\}$
        while (receive-state($k$, $g$, $s'$)) do begin
            if $[h_1(s'), h_2(s')] \notin H_i$ do begin
                $H_i = H_i \cup \{h_1(s'), h_2(s')\}$
                $F_i$.add($s'$)
                $E_i = E_i \cup \{s'\}$
            end
            send-id($k$, $g$, id($s'$))
        end
    end
end
```

Fig. 5. Parallel state space generation algorithm for node i

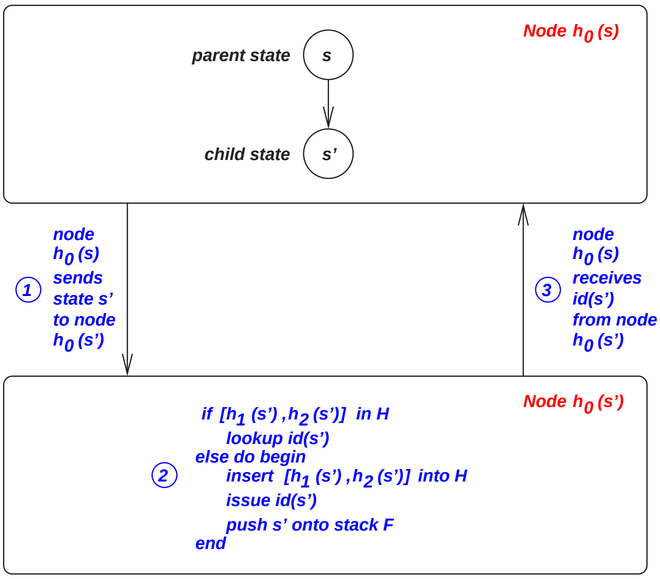


Fig. 6. Steps required to identify child state s' of parent s.

must receive incoming states via matching calls to receive-state(k, g, s) where k is the sender node. If they are not already present, the remote processor adds the incoming states to both the remote state hash table and FIFO queue.

For the purpose of constructing the state graph, states are identified by a pair of integers (i, j) where $i = h_0(s)$ is the node number of the host processor and j is the local state sequence number. As in the sequential case, the index j can be stored in the state hash table of node i. However, a node will not be aware of the state identity numbers of non-local successor states. Therefore, when a node receives a state it returns its identity to the sender by calling send-id(k, g, h) where k is the sender, g is the identity of the parent state and h is the identity of the received state. The identity is received by the original sender via a call to receive-id(g, h). Fig. 6 summarises the main steps that take place to identify and process each child s' of state s in the case that $h_0(s) \neq h_0(s')$.

In practice, it is inefficient to implement the communication as detailed in Fig. 5 and Fig. 6, since the network rapidly becomes overloaded with too many short messages. Consequently state and identity messages are buffered and sent in large blocks. In order to avoid starvation and deadlock, nodes that have very few states left in their FIFO queue or are idle broadcast a message to other nodes requesting them to flush their outgoing message buffers.

The algorithm terminates when all the F_i are empty and there are no outstanding state or identity messages. The problem of determining when these conditions are satisfied across a distributed set of processes is a non-trivial problem. From the several distributed termination algorithms surveyed in [38], we have chosen to use Dijkstra's circulating probe algorithm [39].

Reliability Using the parallel algorithm, two distinct states s_1 and s_2 will be mistakenly classified as identical states if and only if $h_0(s_1) = h_0(s_2)$ and $h_1(s_1) = h_1(s_2)$ and $h_2(s_1) = h_2(s_2)$. Since h_0, h_1 and h_2 are independent functions, the reliability of the parallel algorithm is essentially the same as that of the sequential algorithm with a large hash table of Nr rows, giving a state omission probability of

$$q = \frac{n^2}{Nr2^{b+1}}. \tag{28}$$

Space Complexity In the parallel algorithm, each node supports a hash table with r rows. This requires a total of Nhr bytes of storage. The total amount of space required for the dynamic storage of n states remains the same as for the sequential version, i.e. $(b+s)n/8$ bytes. Thus the total memory requirement across all nodes is given by:

$$M = Nhr + n(b+s)/8.$$

4.2 Parallel Disk-based Steady State Solution

Having generated the state space and state graph, the next challenge in performance analysis is usually to find the long run proportion of time the system spends in each of its states. The state graph maps directly onto a continuous time Markov chain which can then be solved for its steady-state distribution, according to Eq. (2).

Since the resources of a single workstation are usually inadequate to tackle the solution of large models (e.g. simply storing the solution vector of a system with 100 million states requires 800MB memory), we explore distributed out-of-core techniques which leverage the compute power, memory and disk space of several processors.

Scalable Numerical Methods A broad spectrum of sequential solution techniques are available for solving steady-state equations [40]. These include classical iterative methods, Krylov subspace techniques and decomposition-based techniques. Many of these algorithms are unsuited to distributed or parallel implementation, however, since they rely on the so-called "Gauss-Seidel effect" to accelerate convergence. This effect occurs when newly updated steady-state vector elements are used in the calculation of other vector elements within the same iteration. In the case of sparse matrices, this sequential dependency can be alleviated by using multi-coloured ordering schemes which allow parallel computation of unrelated vector elements in phases; however, finding such orderings

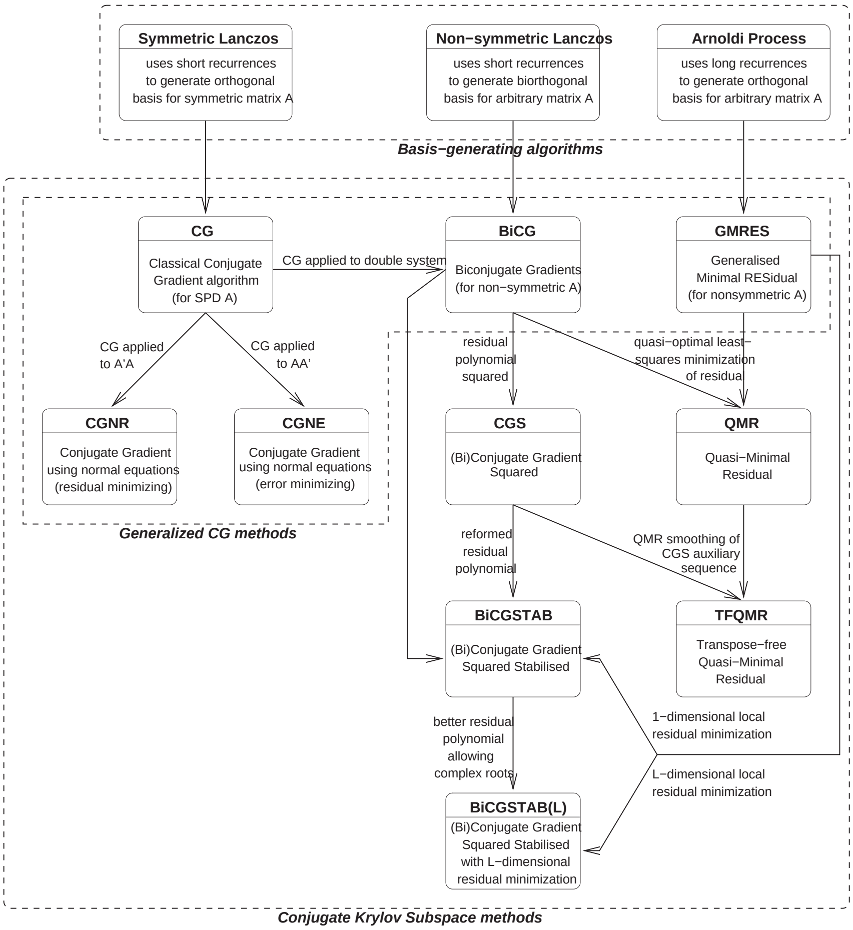

Symmetric Lanczos
uses short recurrences to generate orthogonal basis for symmetric matrix A
Non−symmetric Lanczos
uses short recurrences to generate biorthogonal basis for arbitrary matrix A
Arnoldi Process
uses long recurrences to generate orthogonal basis for arbitrary matrix A
Basis−generating algorithms
CG
Classical Conjugate Gradient algorithm (for SPD A)
CG applied to double system
BiCG
Biconjugate Gradients (for non−symmetric A)
GMRES
Generalised Minimal RESidual (for nonsymmetric A)
CG applied to A'A
CG applied to AA'
residual polynomial squared
quasi−optimal least−squares minimization of residual
CGNR
Conjugate Gradient using normal equations (residual minimizing)
CGNE
Conjugate Gradient using normal equations (error minimizing)
CGS
(Bi)Conjugate Gradient Squared
QMR
Quasi−Minimal Residual
Generalized CG methods
reformed residual polynomial
QMR smoothing of CGS auxiliary sequence
BiCGSTAB
(Bi)Conjugate Gradient Squared Stabilised
TFQMR
Transpose−free Quasi−Minimal Residual
1−dimensional local residual minimization
better residual polynomial allowing complex roots
L−dimensional local residual minimization
BiCGSTAB(L)
(Bi)Conjugate Gradient Squared Stabilised with L−dimensional residual minimization
Conjugate Krylov Subspace methods

Fig. 7. An overview of Krylov subspace techniques

is a combinatorial problem of exponential complexity. Consequently obtaining suitable orderings for very large matrices is infeasible.

Most classical iterative methods, such as Gauss-Seidel and Successive Overrelaxation (SOR), suffer from this problem. An important exception is the Jacobi method which uses independent updates of vector elements. The Jacobi method is characterised by slow, smooth convergence.

Krylov subspace methods [41] are a powerful class of iterative methods which includes many conjugate gradient-type algorithms. They derive their name from the fact that they generate their iterates using a shifted Krylov subspace associated with the coefficient matrix. They are widely used in scientific computing since they are parameter free (unlike SOR) and exhibit rapid, if somewhat erratic, convergence. In addition, these methods are well suited to parallel implementation because they are based on matrix–vector products, independent vector updates and inner products. Fig. 7 presents a conceptual overview of the most important techniques. The arrows show the relationships between the methods, i.e. how the methods have been generalised from their underlying basis-generating algorithms and also how key concepts have been inherited from one algorithm to the next.

The most recently developed Krylov subspace algorithms (such as CGS [42], BiCGSTAB [43] and TFQMR [44]) are also particularly suited to a disk-based implementation since they access $\mathbf{A}$ in a predictable fashion and do not require multiplication with $\mathbf{A}^T$. Compared to classical iterative methods, however, Krylov subspace techniques have high memory requirements. CGS is often used because it requires the least memory of these methods.

Disk-based Solution Techniques The concept of using magnetic disk as a buffer to store data that is too large to fit into main memory is an idea which originated three decades ago with the development of overlays and virtual memory systems. However, only recently, with the widespread availability of large, cheap, high-bandwidth hard disks has attention been focused on the potential of disks as high-throughput data sources appropriate for use in data-intensive computations.

In [45] and [46], Deavours and Sanders make a compelling case for the potential of disk-based steady-state solution methods for large Markov models. They note that our ability to solve large matrices is limited by the memory required to store a representation of the transition matrix and by the effective rate at which matrix elements can be produced from the encoding. As a general rule, the more compact the representation, the more CPU overhead is involved in retrieving matrix elements. Two common encodings are Kronecker representations and "on-the-fly" methods. Deavours and Sanders estimate the effective data production rate of Kronecker and "on-the-fly" methods as being 2 MB/s and 440 KB/s respectively on their 120 MHz HP C110 workstation. Other published results show that an implementation of a state-of-the-art Kronecker technique running on a 450 MHz Pentium-II workstation yields an effective data production rate of around 2.5 MB/s [27].

At the same time, modern workstation disks are capable of sustaining data transfer rates in excess of 20 MB/s. This suggests that it would be worthwhile to store the transition matrix on disk, given that enough disk space is available and given that we can apply an iterative solution method that accesses the transition matrix in a predictable way. Such an approach has the potential to produce data faster than both Kronecker and on-the-fly methods, without any of the structural restrictions inherent in Kronecker methods.

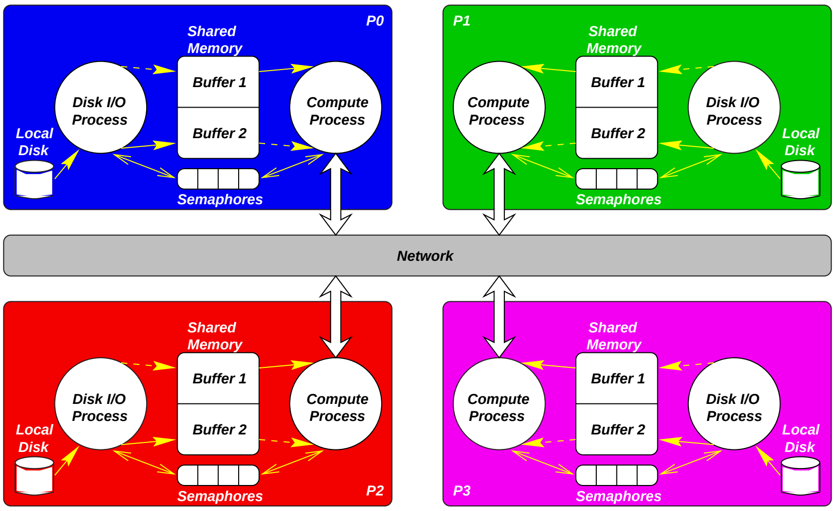


Fig. 8. Distributed disk-based solver architecture

Deavours and Sanders demonstrate the effectiveness of this approach by devising a sequential disk-based solution tool which makes use of two cooperating processes. One of the processes is dedicated to reading disk data while the other performs computation using a Block Gauss-Seidel algorithm, thus allowing for the overlap of disk I/O and computation. The processes communicate using semaphores and shared memory. The advantage of using Block Gauss-Seidel is that diagonal matrix blocks can be read from disk once, be cached in memory and then reused several times.

The memory required by the disk-based approach is small – besides the shared memory buffers, space is only required for the solution vector itself. This enables the solution of extremely large models with over 10 million states and 100 million non-zero entries on a HP C110 workstation with 128MB RAM and 4GB of disk space in just over 5 hours.

Kwiatkowska and Mehmood reduce the memory requirements of disk-based methods even further by proposing a block-based Gauss-Seidel method which uses disk to store blocks of the steady-state vector as well as matrix blocks [47, 48]. In this way, a model of a manufacturing system with 133 million states is solved on a single PC in 13 days and 9 hours.

Parallel disk-based solver architectures have also been implemented with some success. Fig. 8 shows the architecture proposed in [49]. Each node has two processes: a *Disk I/O* process dedicated to reading matrix elements from a local disk, and a *Compute* process which performs the iterations using a Jacobi or CGS-based matrix–vector multiply kernel. The processes share two data buffers located in shared memory and synchronise via semaphores. Together the processes operate as a classical producer-consumer system, with the disk I/O process filling one shared memory buffer while the compute process consumes data from the other.

Bell and Haverkort apply a similar architecture in solving a 724 million state Markov chain model on a 26 node PC cluster in 16 days [50].

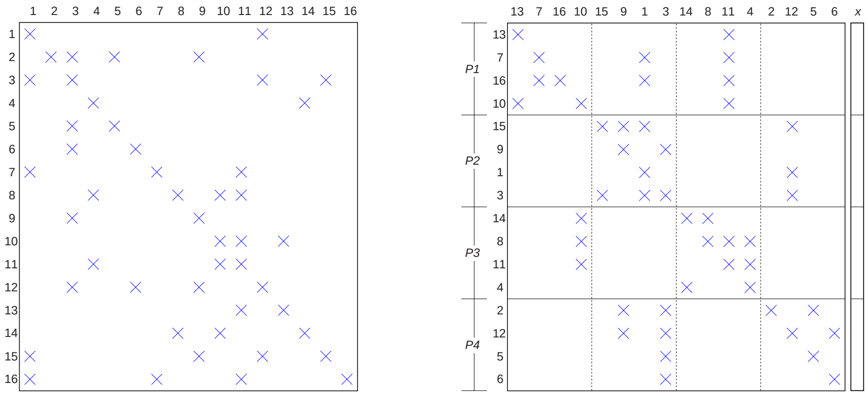


Fig. 9. A 16 × 16 non-symmetric sparse matrix (left), with corresponding 4-way hypergraph partition (right) and corresponding partitions of the vector

Hypergraph Partitioning Any distributed solution scheme involves partitioning the sparse matrix and vector elements across the processors. Such schemes necessitate the exchange of data (vector elements and possibly partial sums) after every iteration in the solution process. The objective in partitioning the matrix is to minimise the amount of data which needs to be exchanged while balancing the computational load (as given by the number of non-zero elements assigned to each processor).

Hypergraph partitioning is an extension of graph partitioning. Its primary application to date has been in VLSI circuit design, where the objective is to cluster pins of devices such that interconnect is minimised. It can also be applied to the problem of allocating the non-zero elements of sparse matrices across processors in parallel computation [51].

Formally, a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined by a set of vertices $\mathcal{V}$ and a set of nets (or hyperedges) $\mathcal{N}$, where each net is a subset of the vertex set $\mathcal{V}$ [51]. In the context of a row-wise decomposition of a sparse matrix $\mathbf{A}$, matrix row i $(1 \leq i \leq n)$ is represented by a vertex $v_i \in \mathcal{V}$ while column j $(1 \leq j \leq n)$ is represented by net $N_j \in \mathcal{N}$. The vertices contained within net N_j correspond to the row numbers of the non-zero elements within column j, i.e. $v_i \in N_j$ if

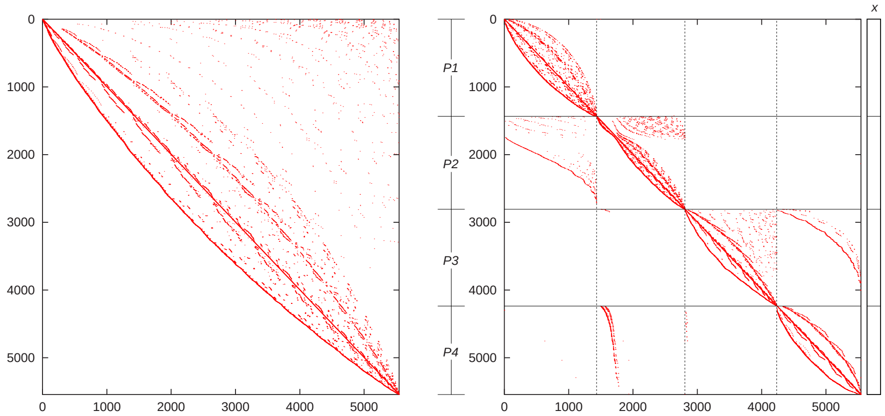


Fig. 10. Transposed transition matrix (left) and corresponding hypergraph-partitioned matrix (right)

and only if $a_{ij} \neq 0$. The weight of vertex i is given by the number of non-zero elements in row i, while the weight of a net is its contribution to the edge cut, which is defined as one less than the number of different partitions spanned by that net. The overall objective of a hypergraph sparse matrix partitioning is to minimise the sum of the weights of the cut nets while maintaining a balance criterion. A column-wise decomposition is achieved in an analogous fashion.

The matrix on the right of Fig. 9 shows the result of applying hypergraph-partitioning to the matrix on the left in a four-way row-wise decomposition. Although the number of off-diagonal non-zeros is 18 the number of vector elements which must be transmitted between processors during each matrix–vector multiplication (the communication cost) is 6. This is because the hypergraph partitioning algorithms not only aim to concentrate the non-zeros on the diagonals but also strive to line up the off-diagonal non-zeros in columns. The edge cut of the decomposition is also 6, and so the hypergraph partitioning edge cut metric exactly quantifies the communication cost. This is a general property and one of the key advantages of using hypergraphs – in contrast to graph partitioning, where the edge cut metric merely approximates communication cost. Optimal hypergraph partitioning is NP-complete but there are a small number of hypergraph partitioning tools which implement fast heuristic algorithms, for example PaToH [51], hMeTiS [52] and Par*k*way [53].

Fig. 10 shows the application of hypergraph partitioning to a (transposed) generator matrix. Statistics about the communication associated with this decomposition for a single matrix–vector multiplication are presented in Tab. 2. We see that around 90% of the non-zero elements allocated to each processor are local, i.e. they are multiplied with vector elements that are stored locally. The

proc-	non-	local	remote	reused
essor	zeros	%	%	%
1	7 022	99.96	0.04	0
2	7 304	91.41	8.59	34.93
3	6 802	88.44	11.56	42.11
4	6 967	89.01	10.99	74.28

	1	2	3	4
1	-	407	-	4
2	3	-	16	181
3	-	-	-	12
4	-	1	439	-

Table 2. Communication overhead (left) and interprocessor communication matrix (right).

remote non-zero elements are multiplied with vector elements that are sent from other processors. However, because the hypergraph decomposition tends to align remote non-zero elements in columns (well illustrated in the 2nd block belonging to processor 4), reuse of received vector elements is good (up to 74%) with correspondingly lower communication overhead. The communication matrix on the right in Tab. 2 shows the number of vector elements sent between each pair of processors during each iteration (e.g. 181 vector elements are sent from processor 2 to processor 4).

4.3 Parallel Computation of Densities and Quantiles of First Passage Time

A rapid response time is an important performance criterion for almost all computer-communication and transaction processing systems. Response time quantiles are frequently specified as key quality of service metrics in Service Level Agreements and industry standard benchmarks such as TPC. Examples of systems with stringent response time requirements include mobile communication systems, stock market trading systems, web servers, database servers, flexible manufacturing systems, communication protocols and communication networks. Typically, response time targets are specified in terms of quantiles – for example “95% of all text messages must be delivered within 3 seconds”.

In the past, numerical computation of analytical response time densities has proved prohibitively expensive except in some Markovian systems with restricted structure such as overtake-free queueing networks [54]. However, with the advent of high-performance parallel computing and the widespread availability of PC clusters, direct numerical analysis on Markov and semi-Markov chains has now become a practical proposition.

There are two main methods for computing first passage time (and hence response time) densities in Markov chains: those based on Laplace transforms and their inversion [55, 56] and those based on uniformisation [8, 6]. The former has wider application to semi-Markov processes but is less efficient than uniformisation when restricted to Markov chains.

Numerical Laplace Transform Inversion The key to practical analysis of semi-Markov processes lies in the efficient representation of their general distributions. Without care the structural complexity of the SMP can be recreated within the representation of the distribution functions.

Many techniques have been used for representing arbitrary distributions – two of the most popular being *phase-type distributions* and *vector-of-moments* methods. These methods suffer from, respectively, exploding representation size under composition, and containing insufficient information to produce accurate answers after large amounts of composition.

As all our distribution manipulations take place in Laplace-space, we link our distribution representation to the Laplace inversion technique that we ultimately use. Our tool supports two Laplace transform inversion algorithms, which are briefly outlined below: the Euler technique [57] and the Laguerre method [58] with modifications summarised in [59].

Both algorithms work on the same general principle of sampling the transform function $L(s)$ at n points, $s_1, s_2, \ldots, s_n$ and generating values of $f(t)$ at m user-specified t-points $t_1, t_2, \ldots, t_m$. In the Euler inversion case $n = km$, where k can vary between 15 and 50, depending on the accuracy of the inversion required. In the modified Laguerre case, $n = 400$ and, crucially, is independent of m.

The process of selecting a Laplace transform inversion algorithm is discussed later; however, whichever is chosen, it is important to note that calculating $s_i, 1 \leq i \leq n$ and storing all our distribution transform functions, sampled at these points, will be sufficient to provide a complete inversion. Key to this is that fact that matrix element operations, of the type performed in Eq. (39), (i.e. convolution and weighted sum) do not require any adjustment to the array of domain s-points required. In the case of a convolution, for instance, if $L_1(s)$ and $L_2(s)$ are stored in the form $\{(s_i, L_j(s_i)) \ : \ 1 \leq i \leq n\}$, for $j = 1, 2$, then the convolution, $L_1(s)L_2(s)$, can be stored using the same size array and using the same list of domain s-values, $\{(s_i, L_1(s_i)L_2(s_i)) \ : \ 1 \leq i \leq n\}$.

Storing our distribution functions in this way has three main advantages. Firstly, the function has constant storage space, independent of the distribution-type. Secondly, each distribution has, therefore, the same constant storage requirement even after composition with other distributions. Finally, the function has sufficient information about a distribution to determine the required passage time (and no more).

Summary of Euler Inversion The Euler method is based on the Bromwich contour inversion integral, expressing the function $f(t)$ in terms of its Laplace transform $L(s)$. Making the contour a vertical line $s = a$ such that $L(s)$ has no singularities on or to the right of it gives:

$$f(t) = \frac{2e^{at}}{\pi} \int_0^\infty Re(L(a + iu)) \cos(ut) \, \mathrm{d}u \tag{29}$$

This integral can be numerically evaluated using the trapezoidal rule with step-size $h = \pi/2t$ and $a = A/2t$ (where A is a constant that controls the discretisation error), which results in the nearly alternating series:

$$f(t) \approx f_h(t) = \frac{e^{A/2}}{2t} Re(L(A/2t)) + \frac{e^{A/2}}{2t} \sum_{k=1}^{\infty} (-1)^k Re\left(L\left(\frac{A + 2k\pi i}{2t} \right) \right) \tag{30}$$

Euler summation is employed to accelerate the convergence of the alternating series infinite sum, so we calculate the sum of the first n terms explicitly and use Euler summation to calculate the next m. To give an accuracy of 10^{-8} we set $A = 19.1, n = 20$ and $m = 12$ (compared with $A = 19.1$, $n = 15$ and $m = 11$ in [57]).

Summary of Laguerre Inversion The Laguerre method [58] makes use of the Laguerre series representation:

$$f(t) = \sum_{n=0}^{\infty} q_n l_n(t) \qquad : t \geq 0 \tag{31}$$

where the Laguerre polynomials l_n are given by:

$$l_n(t) = \left(\frac{2n - 1 - t}{n} \right) l_{n-1}(t) - \left(\frac{n-1}{n} \right) l_{n-2}(t) \tag{32}$$

starting with $l_0 = e^{t/2}$ and $l_1 = (1 - t)e^{t/2}$, and:

$$q_n = \frac{1}{2\pi r^n} \int_0^{\pi} Q(re^{iu}) e^{-iru} \, \mathrm{d}u \tag{33}$$

where $r = (0.1)^{4/n}$ and $Q(z) = (1 - z)^{-1} L((1 + z)/2(1 - z))$.

The integral in the calculation of q_n can be approximated numerically by the trapezoidal rule, giving:

$$q_n \approx \bar{q}_n = \frac{1}{2nr^n} \left(Q(r) + (-1)^n Q(-r) + 2 \sum_{j=1}^{n-1} (-1)^j Re\left(Q(re^{\pi j i/n}) \right) \right) \tag{34}$$

As described in [59], the Laguerre method can be modified by noting that the Laguerre coefficients q_n are independent of t. This means that if the number of trapezoids used in the evaluation of q_n is fixed to be the same for every q_n (rather than depending on the value of n), values of $Q(z)$ (and hence $L(s)$) can be reused after they have been computed. Typically, we set $n = 200$. In order to achieve this, however, the scaling method described in [58] must be used to ensure that the Laguerre coefficients have decayed to (near) 0 by $n = 200$. If this can be accomplished, the inversion of a passage time density for any number of t-values can be achieved at the fixed cost of calculating 400 truncated summations of the type shown in Eq. (39). This is in contrast to the Euler method, where the number of truncated summations required is a function of the number of points at which the value of $f(t)$ is required.

Iterative Passage-Time Analysis for SMPs Passage-time analysis in semi-Markov processes involves the solution of a set of linear equations in complex variables. In [60], we set out an efficient iterative approach to passage time calculation and proved its convergence to the analytic passage time distribution. The algorithm has since been implemented and is used to calculate semi-Markov passage times in the SMARTA tool (described below).

Recall the semi-Markov process, $Z(t)$, of Sect. 2.2, where $N(t)$ is the number of state transitions that have taken place by time t. We formally define the rth transition first passage time to be:

$$P_{i\boldsymbol{j}}^{(r)} = \inf\{u > 0 : Z(u) \in \boldsymbol{j}, N(u) > 0 \mid N(u) \leq r, Z(0) = i\} \tag{35}$$

which is the time taken to enter a state in $\boldsymbol{j}$ for the first time having started in state i at time 0 and having undergone up to r state transitions[1]. $P_{i\boldsymbol{j}}^{(r)}$ is a random variable with associated Laplace transform, $L_{i\boldsymbol{j}}^{(r)}(s)$. $L_{i\boldsymbol{j}}^{(r)}(s)$ is, in turn, the ith component of the vector:

$$\mathbf{L}_{\boldsymbol{j}}^{(r)}(s) = (L_{1\boldsymbol{j}}^{(r)}(s), L_{2\boldsymbol{j}}^{(r)}(s), \ldots, L_{N\boldsymbol{j}}^{(r)}(s)) \tag{36}$$

representing the passage time for terminating in $\boldsymbol{j}$ for each possible start state. This vector may be computed as:

$$\mathbf{L}_{\boldsymbol{j}}^{(r)}(s) = \mathbf{U}(\mathbf{I} + \mathbf{U}' + \mathbf{U}'^{2} + \cdots + \mathbf{U}'^{(r-1)})\,\mathbf{e}_{\boldsymbol{j}} \tag{37}$$

where $\mathbf{U}$ is a matrix with elements $u_{pq} = r_{pq}^{*}(s)$ and $\mathbf{U}'$ is a modified version of $\mathbf{U}$ with elements $u'_{pq} = \delta_{p \notin \boldsymbol{j}}\, u_{pq}$, where states in $\boldsymbol{j}$ have been made absorbing. We include the initial $\mathbf{U}$-transition in Eq. (37), so as to generate cycle times for cases such as $L_{ii}^{(r)}(s)$ which would otherwise register as 0 if $\mathbf{U}'$ were used instead. The column vector $\mathbf{e}_{\boldsymbol{j}}$ has entries $e_{k\boldsymbol{j}} = \delta_{k \in \boldsymbol{j}}$.

From Eq. (15) and Eq. (35):

$$P_{i\boldsymbol{j}} = P_{i\boldsymbol{j}}^{(\infty)} \quad \text{and thus} \quad L_{i\boldsymbol{j}}(s) = L_{i\boldsymbol{j}}^{(\infty)}(s) \tag{38}$$

We can generalise to multiple source states $\boldsymbol{i}$ using the normalised steady-state vector $\boldsymbol{\alpha}$ of Eq. (21):

$$\begin{aligned} L_{\boldsymbol{ij}}^{(r)}(s) &= \boldsymbol{\alpha}\mathbf{L}_{\boldsymbol{j}}^{(r)}(s) \\ &= \textstyle\sum_{k=0}^{r-1} \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^{k}\,\mathbf{e}_{\boldsymbol{j}} \end{aligned} \tag{39}$$

[1] If there are immediate transitions in the semi-Markov process then we have to use a modified formulation of the passage time and iterative passage time definitions [60].

The sum of Eq. (39) can be computed efficiently using sparse matrix–vector multiplications with a vector accumulator, $\boldsymbol{\mu}_r = \sum_{k=0}^{r} \boldsymbol{\alpha}\mathbf{U}\mathbf{U}'^{k}$. At each step, the accumulator (initialised as $\boldsymbol{\mu}_0 = \boldsymbol{\alpha}\mathbf{U}$) is updated with $\boldsymbol{\mu}_{r+1} = \boldsymbol{\alpha}\mathbf{U} + \boldsymbol{\mu}_r\mathbf{U}'$. The worst-case time complexity for this sum is $O(N^2 r)$ versus the $O(N^3)$ of typical matrix inversion techniques. In practice, for a sparse matrix with constant bandwidth (number of non-zeros per row), this can be as low as $O(Nr)$.

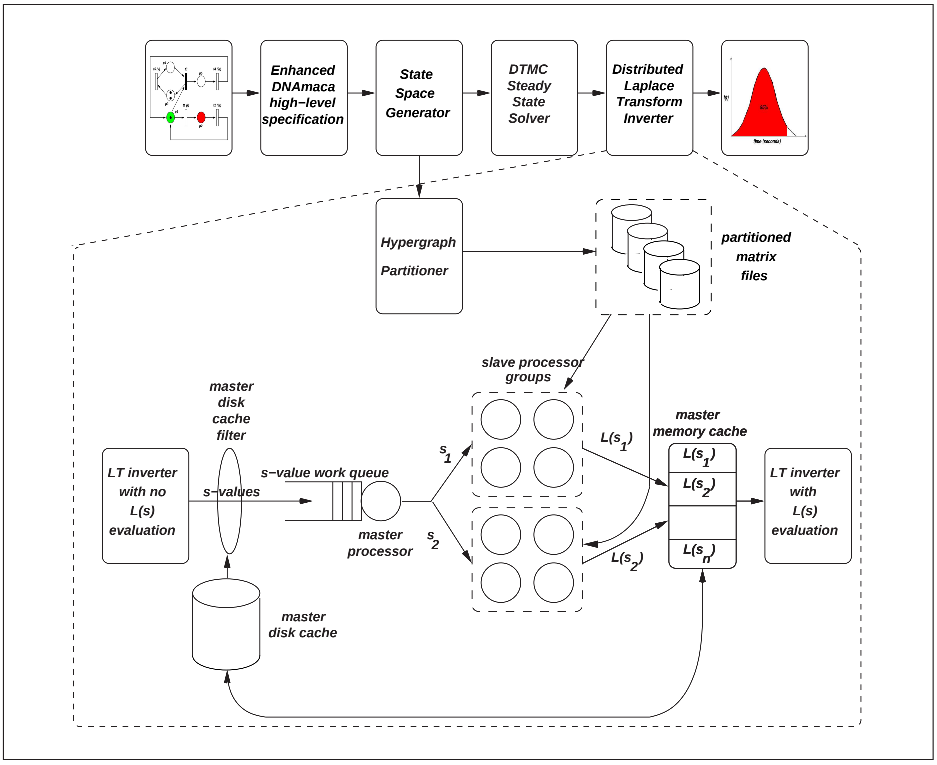


Fig. 11. Parallel hypergraph-based passage time density calculation pipeline

The SMARTA Tool Our iterative passage-time analysis algorithm has been implemented in the SMARTA tool [61], the architecture of which is shown in Fig. 11. The process of calculating a passage time density begins with a high-level model specified in an enhanced form of the DNAmaca interface language [62, 10]. This language supports the specification of stochastic Petri nets, queueing networks and stochastic process algebras. Next, a probabilistic, hash-based state generator [37] uses the high-level model description to produce the transition probability matrix P of the model's embedded Markov chain, the matrices $\mathbf{U}$ and $\mathbf{U}'$, and a list of the initial and target states. Normalised weights for the initial states are determined by the solution of $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$, which is readily done using any of a variety of steady-state solution techniques (e.g. [45, 49]). $\mathbf{U}'$ is then partitioned using a hypergraph partitioning tool (a single partitioning is sufficient since all linear systems to be solved have the same non-zero structure).

Control is then passed to the distributed passage time density calculator, which is implemented in C++ using the Message Passing Interface (MPI) [63] standard. This employs a master-slave architecture with groups of slave processors. The master processor computes in advance the values of s at which it will need to know the value of $L_{ij}(s)$ in order to perform the inversion. This can be done irrespective of the inversion algorithm employed. The s-values are then placed in a global work-queue to which the groups of slave processors make requests.

The highest ranking processor in a group of slaves makes a request to the master for an s-value and is assigned the next one available. This is then broadcast to the other members of the slave group to allow them to construct their columns of the matrix $\mathbf{U}'$ for that specific s. Each processor reads in the columns of the matrix $\mathbf{U}'$ that correspond to its allocated partition into two types of sparse matrix data structure and also reads in the initial source-state weighting vector $\boldsymbol{\alpha}$. *Local* non-zero elements (i.e. those elements in diagonal matrix blocks that will be multiplied with vector elements stored locally) are stored in a conventional compressed sparse column format. *Remote* non-zero elements (i.e. those elements in off-diagonal matrix blocks that must be multiplied with vector elements received from other processors) are stored in an ultrasparse matrix data structure – one for each remote processor – using a coordinate format. Each processor then determines which vector elements need to be received from and sent to every other processor in the group on each iteration, adjusting the row indices in the ultrasparse matrices so that they index into a vector of received elements. This ensures that a minimum amount of communication takes place and makes multiplication of off-diagonal blocks with received vector elements efficient.

For each step in our iterative algorithm, each processor begins by using non-blocking communication primitives to send and receive remote vector elements, while calculating the product of local matrix elements with locally stored vector elements. The use of non-blocking operations allows computation and communication to proceed concurrently on parallel machines where dedicated network hardware supports this effectively. The processor then waits for the completion of non-blocking operations (if they have not already completed) before multiplying received remote vector elements with the relevant ultrasparse matrices and adding their contributions to the local vector-matrix product cumulatively.

Once the calculations of a slave group are deemed to have converged, the result is returned to the master by the highest-ranking processor in the group and cached. When all results have been computed and returned for all required values of s, the final Laplace inversion calculations are made by the master, resulting in the required t-points.

5 Application examples

We demonstrate our analysis techniques on three application examples: a GSPN model of a communications protocol, a SM-SPN model of a voting system and a PEPA model of an active badge system.

5.1 Courier Protocol Model

Description The GSPN shown in Fig. 12 (originally presented in [64]) models the ISO Application, Session and Transport layers of the Courier sliding-window communication protocol. Data flows from a sender ($p1$ to $p26$) to a receiver ($p27$ to $p46$) via a network. The sender's transport layer fragments outgoing data packets; this is modelled as two paths between $p13$ and $p35$. The path via $t8$ carries all fragments before the last one through the network to $p33$. Acknowledgements for these fragments are sent back to the sender (as signalled by the arrival of a token on $p20$), but no data is delivered to the higher layers on the receiver side. The path via $t9$ carries the last fragment of each message block. Acknowledgements for these fragments are generated and a data token is delivered to higher receiver layers via $t27$.

The average number of data packets sent is determined by the ratio of the weights on the immediate transitions $t8$ and $t9$. This ratio, known as the fragmentation ratio, is given by $q1 : q2$ (where $q1$ and $q2$ are the weights associated with transitions $t8$ and $t9$ respectively). Thus, this number of data packets is geometrically distributed, with parameter $q1/(q1 + q2)$. Here we use a fragmentation ratio of one.

The transport layer is further characterised by two important parameters: the sliding window size n ($p14$) and the transport space m ($p17$). Different values of m and n yield state spaces of various sizes. The transition rates $r1, r2, \dots, r10$ have the same relative magnitudes as those obtained by benchmarking a working implementation of the protocol (see [64]).

State Space Generation We have implemented the state generation algorithm of Fig. 5 on a Fujitsu AP3000 distributed memory parallel computer. Our implementation is written in C++ with support for two popular parallel programming interfaces, viz. the Message Passing Interface (MPI) [65] and the Parallel Virtual Machine (PVM) interface [66]. The generator uses hash tables with $r = 750\,019$ rows per processor and $b = 40$ bit secondary keys. The results were collected using up to 16 processors on the AP3000. Each processor has a 300MHz UltraSPARC processor, 256MB RAM and a 4GB local disk. The nodes run the Solaris operating system and support MPI. They are connected by a high-speed wormhole-routed network with a peak throughput of 65MB/s.

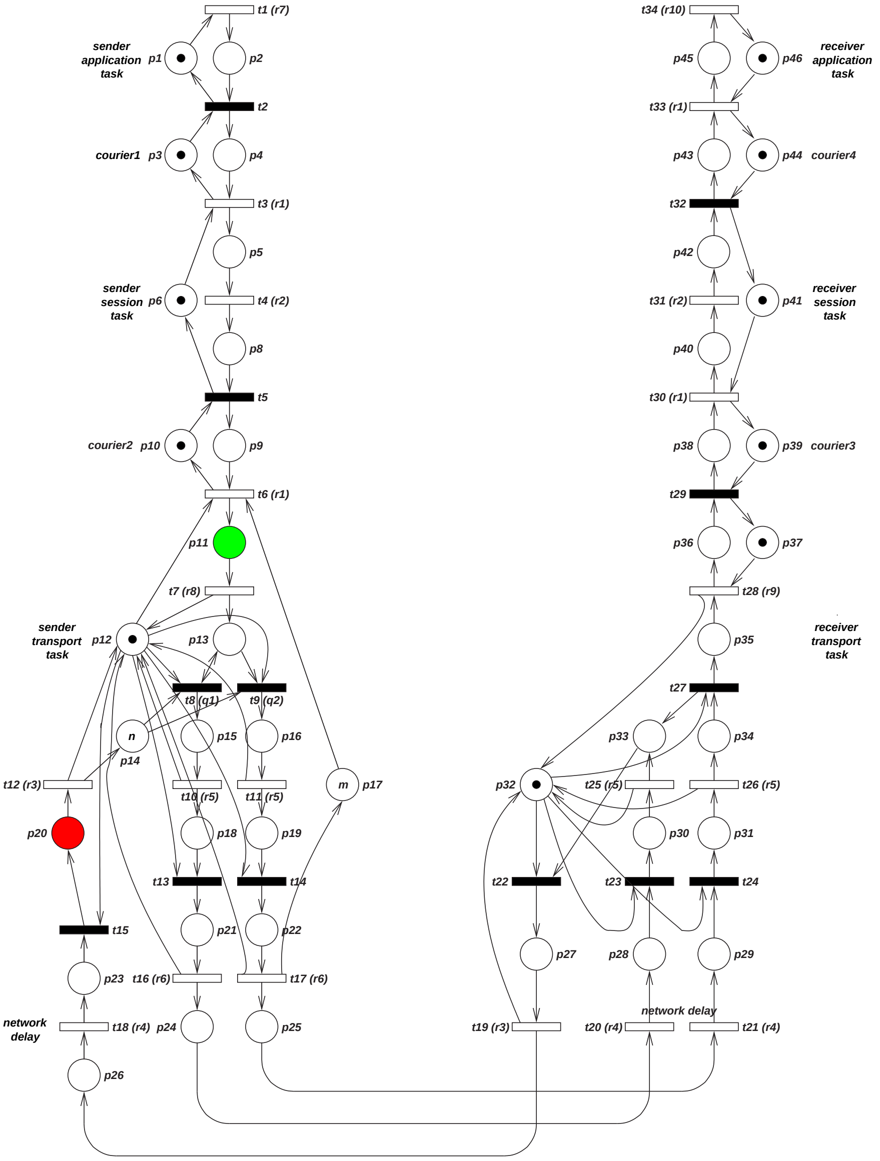


Fig. 12. The Courier Protocol Software Generalised Stochastic Petri net [64].

k	n	a
1	11 700	48 330
2	84 600	410 160
3	419 400	2 281 620
4	1 632 600	9 732 330
5	5 358 600	34 424 280
6	15 410 250	105 345 900
7	39 836 700	286 938 630
8	94 322 250	710 223 930

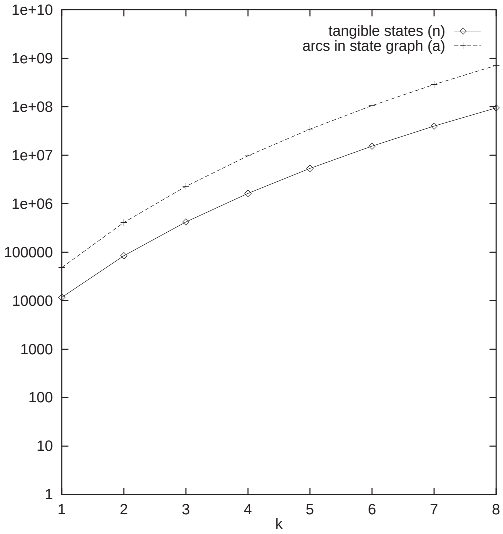


Fig. 13. The number of tangible states (n) and the number of arcs (a) in the state graph of the Courier model for various values of k.

The Courier model features a scaling parameter k (corresponding to the sliding window size) which we will vary to produce state graphs of different sizes (see Fig. 13).

The graph on the left of Fig. 14 shows the distributed run-time taken to explore Courier state spaces of various sizes (up to $k = 6$) using 1, 2, 4, 8, 12 and 16 processors on the AP3000. Each observed value is calculated as the mean of four runs. The $k = 5$ state space (5 358 600 states) can be generated on a single processor in 16 minutes 20 seconds; 16 processors require only 89 seconds. The $k = 6$ state space (15 410 250 states) can be generated on a single processor in 51 minutes 45 seconds; 16 processors require just 267 seconds.

The corresponding speedups for the cases $k = 1, 2, 3, 4, 5, 6$ are shown in the graph on the right of Fig. 14. For $k = 6$ using 16 processors, we observe a speedup of 11.65, giving an efficiency of 73%.

Memory utilisation is low – a single processor generating the $k = 5$ state space uses a total of 91MB (17.4 bytes per state), while the $k = 6$ state space requires 175MB (11.6 bytes per state). This is far less than the 94 bytes per state (45 16-bit integers plus a 32-bit unique state indentifier) that would be required by a straightforward exhaustive implementation.

Moving beyond the maximum state space size that can be generated on a single processor, on 16 processors we find that the $k = 7$ state space can be generated in around 10 minutes while just under 26 minutes are required to generate the $k = 8$ state space (with 94 322 250 states and 710 223 930 arcs).

		k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
$p = 1$	Jacobi time (s)	33.647	278.18	1506.4	5550.3				
	Jacobi iterations	4925	4380	4060	3655				
	CGS time (s)	2.1994	21.622	163.87	934.27	29134			
	CGS iterations	60	81	106	129	157			
	Memory/node (MB)	20.3	22.1	30.5	60.8	154.0			
$p = 2$	Jacobi time (s)	29.655	176.62	1105.7	4313.6				
	Jacobi iterations	4925	4380	4060	3655				
	CGS time (s)	1.6816	13.119	93.28	509.90	7936.9			
	CGS iterations	57	84	107	131	148			
	Memory/node (MB)	20.2	21.1	25.45	41.2	89.7			
$p = 4$	Jacobi time (s)	25.294	148.45	627.96	3328.3				
	Jacobi iterations	4925	4380	4060	3655				
	CGS time (s)	1.2647	8.4109	58.302	322.50	1480.5			
	CGS iterations	60	80	108	133	159			
	Memory/node (MB)	20.1	20.6	22.9	31.4	57.5			
$p = 8$	Jacobi time (s)	38.958	140.06	477.02	1780.9	6585.4			
	Jacobi iterations	4925	4380	4060	3655	3235			
	CGS time (s)	1.4074	6.0976	39.999	204.46	934.76	4258.7		
	CGS iterations	61	82	109	132	155	171		
	Memory/node (MB)	20.0	20.3	21.7	26.5	41.4	81.6		
$p = 12$	Jacobi time (s)	32.152	133.58	457.23	1559.0	6329.2	11578	72202	
	Jacobi iterations	4925	4380	4060	3655	3235	2325	2190	
	CGS time (s)	1.4973	5.9345	34.001	157.73	852.53	2579.6	21220	
	CGS iterations	58	83	104	129	156	189	180	
	Memory/node (MB)	20.0	20.3	21.3	24.9	36.1	66.2	99.7	
$p = 16$	Jacobi time (s)	41.831	125.68	506.31	1547.9	5703.4	11683	32329	
	Jacobi iterations	4925	4380	4060	3650	3235	2325	2190	
	CGS time (s)	3.3505	7.1101	31.322	134.48	577.68	2032.5	13786	141383
	CGS iterations	60	91	104	132	146	173	179	213
	Memory/node (MB)	20.0	20.2	21.0	24.1	33.4	58.5	79.8	161

Table 3. Real time in seconds required for the distributed solution of the Courier model.

Steady-state Analysis Tab. 3 presents the execution time (defined as maximum processor run-time) in seconds required for the distributed disk-based solution of models using the CGS and Jacobi methods. The models range in size from $k = 1$ (11 700 states) to $k = 7$ (39.8 million states) and runs are conducted on 1, 2, 4, 8, 12 and 16 processors. The number of iterations for convergence and memory use per processor are also shown.

Fig. 15 compares the convergence of the Jacobi method with that of the CGS algorithm for the $k = 4$ case in terms of the number of matrix multiplications performed. As is typical for many models, the Jacobi method begins by converging quickly, but then plateaus, converging very slowly but smoothly. The CGS algorithm, on the other hand, exhibits erratic rapid convergence that improves in a concave fashion.

The largest state space solved is the $k = 8$ case (94 million states) which takes 1 day 15 hours of processing time on 16 processors. The total amount of I/O across all nodes is 3.4TB, with the nodes jointly processing an average of 24MB disk data every second.

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$
λ	74.3467	120.372	150.794	172.011	187.413	198.919	207.690	214.477
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705	0.02825	0.02917
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027	0.94808	0.94638
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268	0.02368	0.02445
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80196	0.79320	0.78642
$P_{transp1}$	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632	0.40102	0.38145
$P_{transp2}$	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365	0.40835	0.38871

Table 4. Courier Protocol performance measures in terms of the transport window size k.

Using the steady state vector, it is straightforward to derive some simple resource-based performance measures, as shown in Tab. 4. The most important is λ, the data throughput rate, which is given by the throughput of transition $t21$. Other measures yield task utilizations. In particular, $P_{transp1} = Pr\{p12 \text{ is marked}\} = Pr\{\text{transport task 1 is idle}\}$. Similarly we define $P_{transp2}$ for $p32$, P_{sess1} and P_{sess2} using $p6$ and $p41$, and P_{send} and P_{recv} using $p1$ and $p46$.

First Passage Time Analysis We now apply our iterative passage-time analysis technique to determine the end-to-end response time from the initiation of a transport layer transmission to the arrival of the corresponding acknowledgement packet. Consequently we choose as source markings those markings for which $M(p_{11}) > 0$, and as destination markings those markings for which $M(p_{20}) > 0$. This approach works easily for a sliding window size of $n = 1$ since there can be only one outstanding unacknowledged packet. Naturally, if we wished to calculate the response time for sliding window sizes greater than one, we would need to augment the state vector used to describe markings to track the progress of a particular token through the Petri net.

The underlying reachability graph contains 29 010 markings, 11 700 of which are tangible and 17 310 of which are vanishing. There are 7 320 source markings and 1 680 destination markings. Fig. 16 shows the resulting numerical response time density. The median (50% quantile) and 95% quantile transmission times are also given. Once again the numerical results are compared against a simulation, and agreement is excellent.

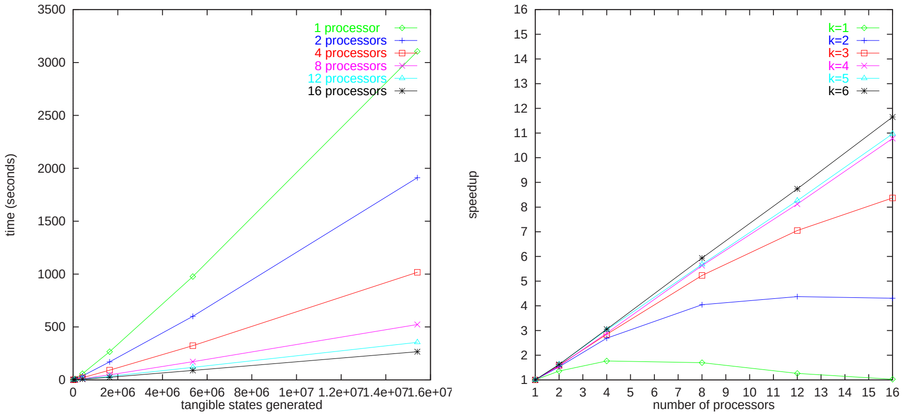


Fig. 14. Real time taken to generate Courier state spaces up to $k = 6$ using the original algorithm on 1, 2, 4, 8, 12 and 16 processors (left), and the resulting speedups for $k = 1, 2, 3, 4, 5$ and 6 (right).

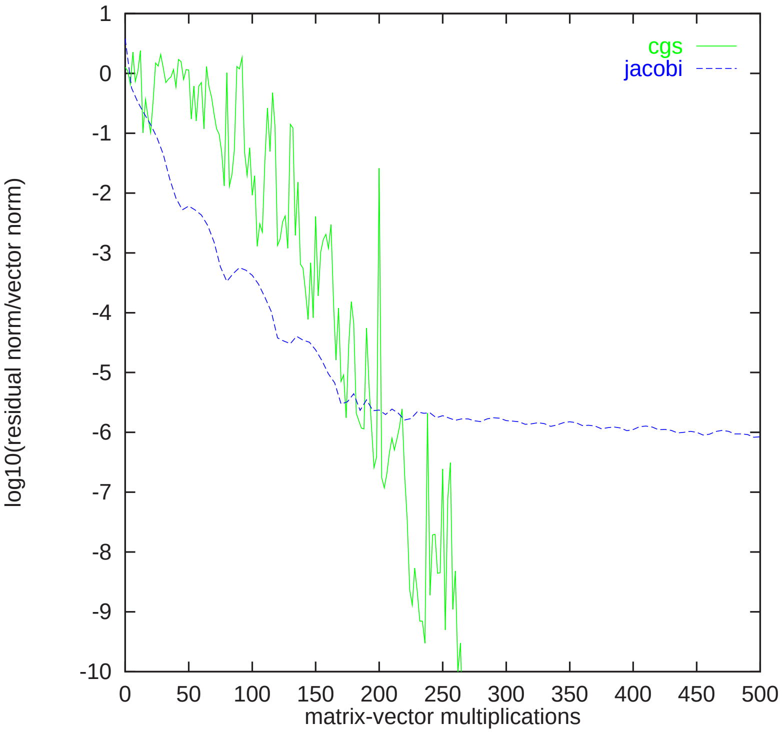


Fig. 15. Jacobi and CGS convergence behaviour for the Courier model with $k = 4$

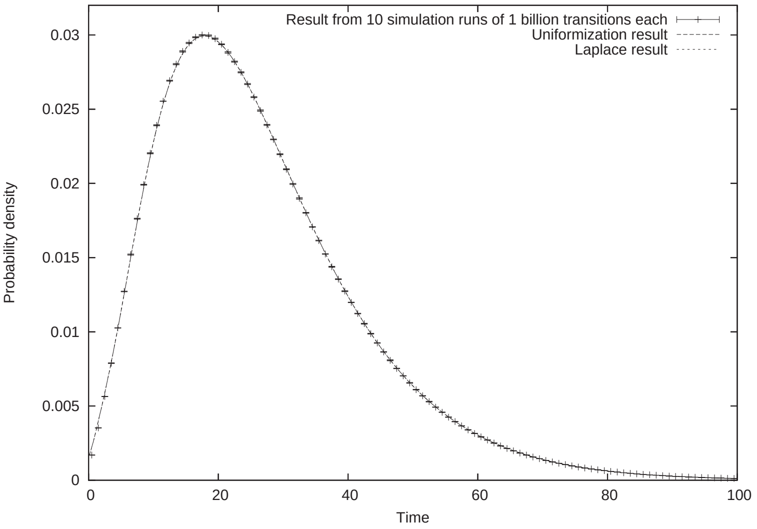


Fig. 16. Numerical and simulated response time densities for time taken from the initiation of a transport layer transmission (i.e. those markings for which $M(p_{11}) > 0$) to the arrival of an acknowledgement packet (i.e. those markings for which $M(p_{20}) > 0$). The median response time (50% quantile) is 0.0048 seconds, and the 95% quantile is 0.0114 seconds.

For this example, our Laguerre scaling algorithm selected a damping parameter of $\sigma = 0.008$. A single slave (a 1.4GHz Athlon processor with 256MB RAM) required 24 minutes 15 seconds to calculate the 200 points plotted on the numerical passage time density graph. Using 8 slave PCs with the same configuration decreased the required time to just 3 minutes 23 seconds (corresponding to an efficiency of 96%). 16 slave PCs required 2 minutes 17 seconds (72% efficiency). These results reflect the excellent scalability of our approach.

5.2 Voting Model

Description Fig. 17 represents a voting system with CC voters, MM polling units and NN central voting servers. In this system, voters cast votes through polling units which in turn register votes with all available central voting units. Both polling units and central voting units can suffer breakdowns, from which there is a soft recovery mechanism. If, however, all the polling or voting units fail, then, with high priority, a failure recovery mode is instituted to restore the system to an operational state.

We demonstrate the SMP passage-time analysis techniques of the previous sections with a large semi-Markov model of a distributed voting system (Fig. 17).

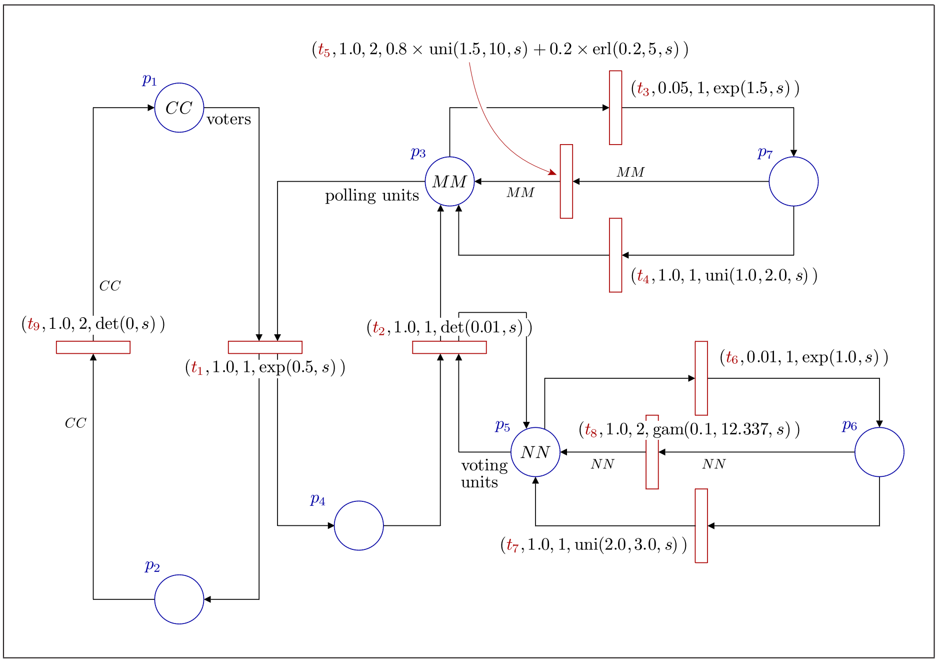

$(t_5, 1.0, 2, 0.8 \times \mathrm{uni}(1.5, 10, s) + 0.2 \times \mathrm{erl}(0.2, 5, s))$
p_1
CC
voters
$(t_3, 0.05, 1, \exp(1.5, s))$
p_3
MM
p_7
MM
polling units
MM
$(t_4, 1.0, 1, \mathrm{uni}(1.0, 2.0, s))$
CC
$(t_9, 1.0, 2, \det(0, s))$
$(t_2, 1.0, 1, \det(0.01, s))$
$(t_1, 1.0, 1, \exp(0.5, s))$
$(t_6, 0.01, 1, \exp(1.0, s))$
CC
p_5
$(t_8, 1.0, 2, \mathrm{gam}(0.1, 12.337, s))$
p_6
NN
voting units
NN
NN
p_4
p_2
$(t_7, 1.0, 1, \mathrm{uni}(2.0, 3.0, s))$

Fig. 17. A semi-Markov stochastic Petri net of a voting system with breakdowns and repairs

The model is specified in a semi-Markov stochastic Petri net (SM-SPN) formalism [67] using an extension of the DNAmaca Markov chain modelling language [62].

The distributions are specified directly as Laplace transforms with certain macros provided for popular distributions (e.g. uniform, gamma, deterministic) and can be made marking dependent by use of the $m(p_i)$ function (which returns the current number of tokens at place, p_i). Support for inhibiting transitions is also provided.

For the voting system, Tab. 5 shows how the size of the underlying SMP varies according to the configuration of the variables CC, MM, and NN.

First Passage Time Analysis The results presented in this section were produced on a Beowulf Linux cluster with 64 dual processor nodes, a maximum of 34 of which can be used by a single job. Each node has two Intel Xeon 2.0GHz processors and 2GB of RAM. The nodes are connected by a Myrinet network with a peak throughput of 250 Mb/s.

Table 5. Different configurations of the voting system and state space generated

System	CC	MM	NN	States
1	60	25	4	106 540
2	100	30	4	249 760
3	125	40	4	541 280
4	150	40	5	778 850
5	175	45	5	1 140 050
6	300	80	10	10 999 140

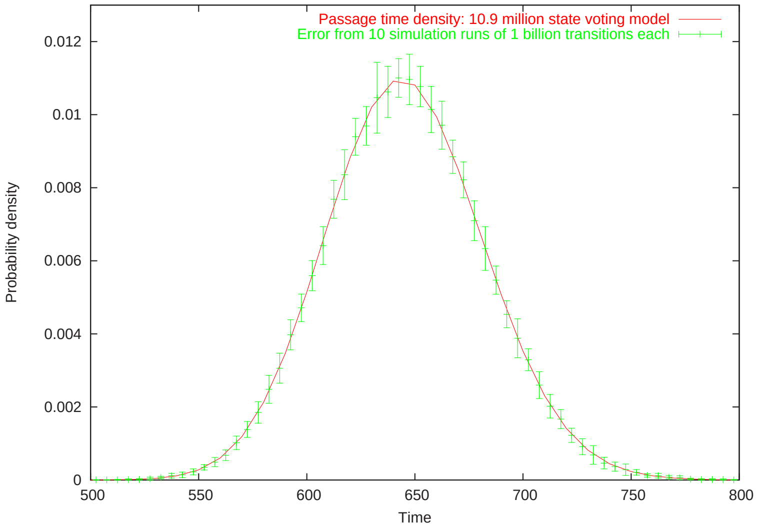


Fig. 18. Analytic and simulated (with 95% confidence intervals) density for the time taken to process 300 voters in the voting model system 6 (10.9 million states).

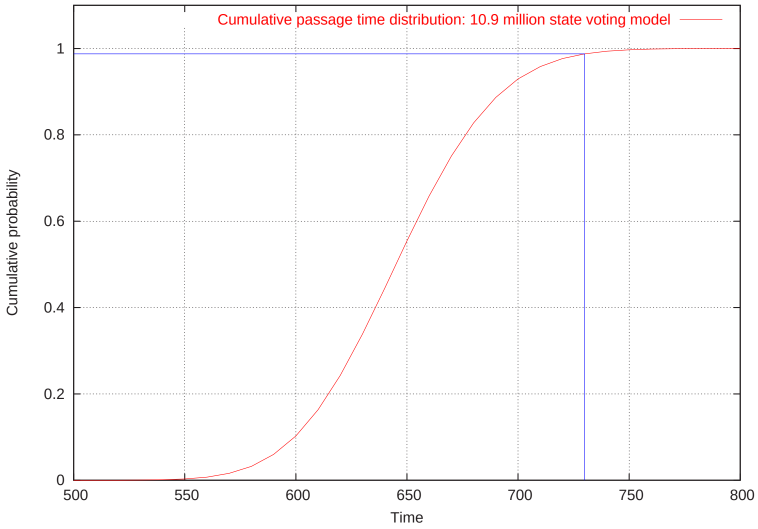


Fig. 19. Cumulative distribution function and quantile of the time taken to process 300 voters in the voting model system 6 (10.9 million states).

We display passage time densities produced by the iterative passage time algorithm and also by simulation to validate those results.

Fig. 18 shows the density of the time taken to process 300 voters (as given by the passage of 300 tokens from place p_1 to p_2) in system 6 of the voting model. Calculation of the analytical density required 15 hours and 7 minutes using 64 slave processors (in 8 groups of 8) for the 31 t-points plotted. Our algorithm evaluated $L_{ij}(s)$ at 1 023 s-points, each of which involved manipulating sparse matrices of rank 10 999 140. The analytical curve is validated against the combined results from 10 simulations, each of which consisted of 1 billion transition firings. Despite this large simulation effort, we still observe wide confidence intervals (probably because of the rarity of source states).

Fig. 19 is a cumulative distribution for the same passage as Fig. 18 (easily obtained by inverting $L_{ij}(s)/s$ from cached values of $L_{ij}(s)$). It allows us to extract response time quantiles, for instance:

$$\mathbb{P}(\text{system 6 can process 300 voters in less than 730 seconds}) = 0.9876$$

5.3 PEPA Active Badge Model

Description In the original active badge model, described in [68], there are 4 rooms on a corridor, all installed with active badge sensors, and a single person who can move from one room to an adjacent room. The sensors are linked to

$$
\begin{aligned}
\mathit{Person}_1 &= (\mathit{reg}_1, r).\mathit{Person}_1 + (\mathit{move}_2, m).\mathit{Person}_2 \\
\mathit{Person}_i &= (\mathit{move}_{i-1}, m).\mathit{Person}_{i-1} + (\mathit{reg}_i, r).\mathit{Person}_i \\
&\quad + (\mathit{move}_{i+1}, m).\mathit{Person}_{i+1} \\
&\qquad : 1 < i < N \\
\mathit{Person}_N &= (\mathit{move}_{N-1}, m).\mathit{Person}_{N-1} + (\mathit{reg}_N, r).\mathit{Person}_N \\[1ex]
\mathit{Sensor}_i &= (\mathit{reg}_i, \top).(\mathit{rep}_i, s).\mathit{Sensor}_i \quad : 1 \leq i \leq N \\[1ex]
\mathit{Dbase}_i &= \textstyle\sum_{j=1}^{N} (\mathit{rep}_j, \top).\mathit{Dbase}_j \quad : 1 \leq i \leq N \\[1ex]
\mathit{Sys} &= \textstyle\prod_{j=1}^{M} \mathit{Person}_1 \underset{\mathit{Reg}}{\bowtie} \prod_{j=1}^{N} \mathit{Sensor}_j \underset{\mathit{Rep}}{\bowtie} \mathit{Dbase}_1
\end{aligned}
$$

where $\mathit{Reg} = \{\mathit{reg}_i \mid 1 \leq i \leq N\}$ and $\mathit{Rep} = \{\mathit{rep}_i \mid 1 \leq i \leq N\}$

Fig. 20. The PEPA description for the generalised active badge model with N rooms and M people.

a database which records which sensor has been activated last. In the model of Fig. 20, we have M people in N rooms with sensors and a database that can be in one of N states. To maintain a reasonable state space, this is a simple database which does not attempt to keep track of every individual's location; rather it remembers the last movement that was made by any person in the system.

In the model below, Person_i represents a person in room i, Sensor_i is the sensor in room i and Dbase_i is the state of the database. A person in room i can either move to room $i-1$ or $i+1$ or, if they remain there long enough, set off the sensor in room i, which registers its activation with the database.

The first thing to note about such a model is how fast the state space can grow. With M people in N rooms, we already have N^M states just from the different configurations of people in rooms. Then there are 2^N sensor configurations and finally N states that the database can be in, giving us a total of $2^N N^{M+1}$ states. For as few as 3 people and 6 rooms, the example we use, we have a global state space of $82,944$ states.

First Passage Time Analysis We include two passages from the active badge system with 3 people and 6 possible rooms. As the model of Fig. 20 tells us, all 6 people start in room 1 and move out from there.

Fig. 21 shows the density function for the passage representing how long it takes for all 3 people to be together in room 6 for the first time.

It is interesting to observe that it is virtually impossible for all 3 people to end up in room 6, which requires 6 successive *move* transitions from all 3 people for

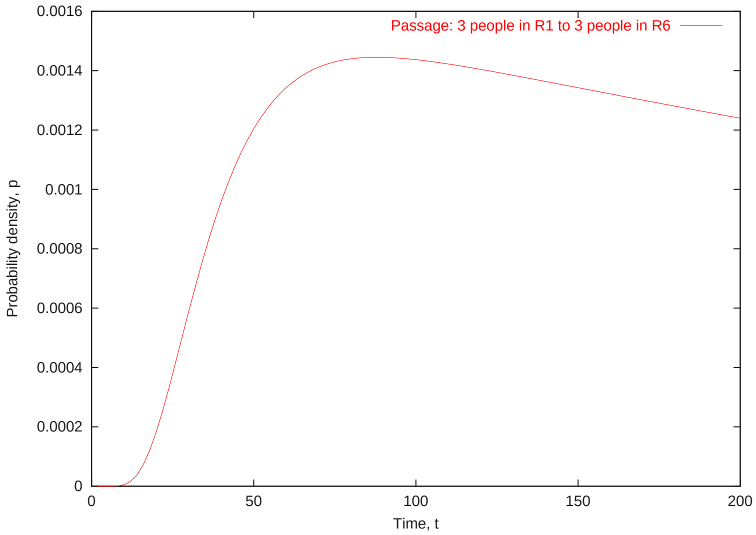


Fig. 21. The passage time density for 3 people starting in room 1 ending up all in room 6.

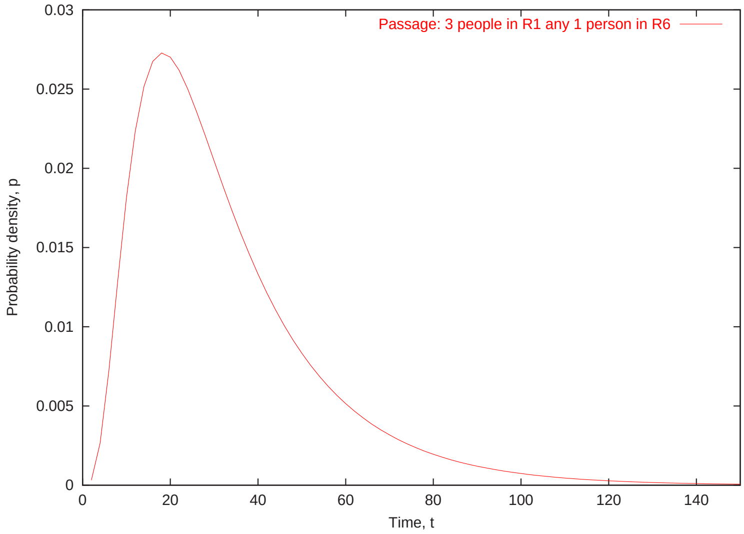


Fig. 22. The passage time density for 3 people starting in room 1 ending up with any one or more of them in room 6.

it to happen at the earliest opportunity, until at least 10 time units have elapsed. After that time, very low probabilities are registered and the distribution clearly has a very heavy tail.

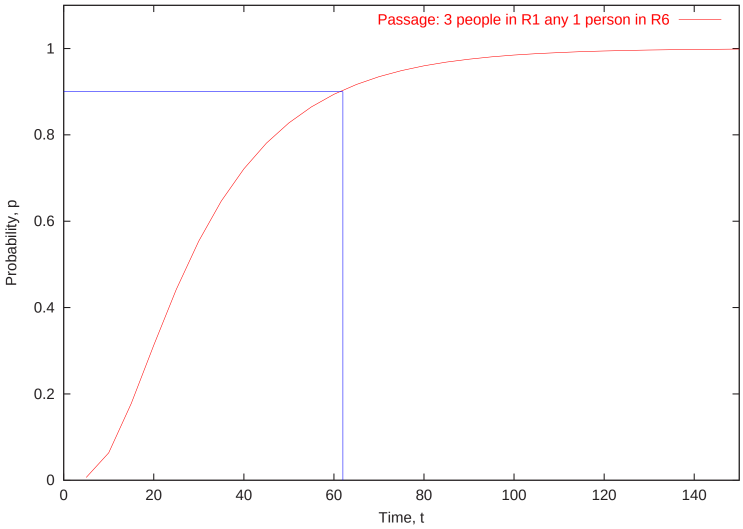


Fig. 23. The cumulative passage time distribution function for 3 people starting in room 1 ending up with any one or more of them in room 6.

The second passage of Fig. 22 shows an equivalent passage time density from the same start point to a terminating condition of at least one person of the three entering room 6. The resulting passage is much less heavy tailed, as this time only a single person has to make it to room 6 before the passage ends.

From these densities, it is a simple matter to construct cumulative distribution functions (the integral of the density function) and obtain quantiles, e.g. the probability that 3 people all reach room 6 by time $t = 150$. Fig. 23 shows the cumulative distribution function (cdf) corresponding to the passage time density of Fig. 22. From this cdf, we can ascertain, for example, that there is a 90% probability that at least one person will have reached room 6 by time $t = 62$.

6 Conclusion and Future Perspectives

Performance analysis of complex systems is a computationally expensive activity. If a model does not have exploitable symmetries or other structure that allows for analytical or numerical shortcuts to be used, then an explicit representation of the state space has to be constructed. This chapter has discussed some state space generation methods and numerical algorithms for steady-state and passage-time analysis of (semi-)Markov models which are scalable across large computing clusters. We have shown that by making use of probabilistic algorithms and efficient distribution strategies (e.g. using hypergraph partitioning), we can subdivide large performance analysis problems in such a way that makes them tractable on individual computer nodes.

An important emerging development with the potential to tackle exceptionally large state spaces is the use of continuous approximations to represent large discrete state spaces. Preliminary efforts to relate this to modelling formalisms have led to continuous state-space translations from SPNs [69] and PEPA [70]. In both cases, repeated structures in the top-level formalism are represented by systems of ordinary differential equations (ODEs) which describe a deterministic trace of behaviour. In certain structural situations [71] the steady-state solution of the ODEs corresponds to the steady-state solution of the underlying Markov chain.

Acknowledgements

The authors would like to thank Nicholas Dingle, Peter Harrison and Aleksandar Trifunovic who contributed hugely to the development of the work presented here and without whom this would not have been possible.

References

1. J. Yang, C. Sar, and D. Engler, "eXplode: a Lightweight, General System for Finding Serious Storage System Errors," in *Proc. 7th Symposium on Operating System Design and Implementation*, (Seattle, WA), pp. 131–146, November 2006.
2. F. Bause and P. Kritzinger, *Stochastic Petri Nets – An Introduction to the Theory.* Wiesbaden, Germany: Verlag Vieweg, 1995.
3. W. Grassman, "Means and variances of time averages in Markovian environments," *European Journal of Operational Research*, vol. 31, no. 1, pp. 132–139, 1987.
4. A. Reibman and K. Trivedi, "Numerical transient analysis of Markov models," *Computers and Operations Research*, vol. 15, no. 1, pp. 19–36, 1988.
5. G. Bolch, S. Greiner, H. Meer, and K. Trivedi, *Queueing Networks and Markov Chains.* Wiley, August 1998.
6. B. Melamed and M. Yadin, "Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes," *Operations Research*, vol. 32, pp. 926–944, July–August 1984.
7. A. Miner, "Computing response time distributions using stochastic Petri nets and matrix diagrams," in *Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, (Urbana-Champaign, IL), pp. 10–19, September 2nd–5th 2003.
8. J. Muppala and K. Trivedi, "Numerical transient analysis of finite Markovian queueing systems," in *Queueing and Related Models* (U. Bhat and I. Basawa, eds.), pp. 262–284, Oxford University Press, 1992.
9. M. Ajmone-Marsan, G. Conte, and G. Balbo, "A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems," *ACM Transactions on Computer Systems*, vol. 2, pp. 93–122, 1984.

10. W. Knottenbelt, *Parallel Performance Analysis of Large Markov Models.* PhD thesis, Imperial College London, February 2000.
11. J. T. Bradley, N. J. Dingle, P. G. Harrison, and W. J. Knottenbelt, "Distributed computation of passage time quantiles and transient state distributions in large semi-Markov models," in *PMEO'03, Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems*, (Nice), p. 281, IEEE Computer Society Press, April 2003.
12. C. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.
13. R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.
14. J. Hillston, *A Compositional Approach to Performance Modelling.* PhD thesis, University of Edinburgh, 1994.
15. M. Rettelbach and M. Siegle, "Compositional minimal semantics for the stochastic process algebra TIPP," in *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, pp. 31–50, Regensberg/Erlangen: Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, July 1994.
16. H. Hermanns and M. Rettelbach, "Syntax, semantics, equivalences and axioms for MTIPP," in *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, Regensberg/Erlangen: Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, July 1994.
17. P. Buchholz, "Markovian Process Algebra: composition and equivalence," in *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, Regensberg/Erlangen: Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, July 1994.
18. M. Bernardo, L. Donatiello, and R. Gorrieri, "Modelling and analyzing concurrent systems with MPA," in *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, pp. 89–106, Regensberg/Erlangen: Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, July 1994.
19. H. Hermanns, U. Herzog, and J. Hillston, "Stochastic process algebras—A formal approach to performance modelling," tutorial, Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK, 1996.
20. B. Strulo, *Process Algebra for Discrete Event Simulation.* PhD thesis, Imperial College, London, October 1993.
21. P. G. Harrison and B. Strulo, "SPADES - a process algebra for discrete event simulation," *Journal of Logic and Computation*, vol. 10, pp. 3–42, January 2000.
22. J. T. Bradley, "Semi-Markov PEPA: Modelling with generally distributed actions," *International Journal of Simulation*, vol. 6, pp. 43–51, January 2005.
23. M. Bravetti, M. Bernardo, and R. Gorrieri, "Towards performance evaluation with general distributions in process algebras," in *CONCUR'98, Proceedings of the 9th International Conference on Concurrency Theory* (D. Sangiorgi and R. de Simone, eds.), vol. 1466 of *Lecture Notes in Computer Science*, pp. 405–422, Springer-Verlag, Nice, September 1998.
24. M. Bravetti and R. Gorrieri, "Interactive generalized semi-Markov processes," in *Process Algebra and Performance Modelling Workshop* (J. Hillston and M. Silva, eds.), pp. 83–98, Centro Politécnico Superior de la Universidad de Zaragoza, Prensas Universitarias de Zaragoza, Zaragoza, September 1999.
25. J. Hillston and M. Ribaudo, "Stochastic process algebras: a new approach to performance modelling," in *Modelling and Simulation of Advanced Computer Systems* (K. Bagchi and G. Zobrist, eds.), ch. 10, pp. 235–256, Gordon Breach, 1998.
26. R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

27. G. Ciardo and A. Miner, "A data structure for the efficient Kronecker solution of GSPNs," in *Proceedings of the 8th International Conference on Petri Nets and Performance Models (PNPM'99)*, (Zaragoza, Spain), pp. 22–31, IEEE Computer Society Press, September 1999.
28. A. Miner and D. Parker, "Symbolic representations and analysis of large probabilistic systems," in *Lecture Notes in Computer Science 2925/2004: Validation of Stochastic Systems*, pp. 296–338, 2004.
29. P. Buchholz and P. Kemper, "Kronceker based matrix representations for large Markov models," in *Lecture Notes in Computer Science 2925/2004: Validation of Stochastic Systems*, pp. 256–295, 2004.
30. M. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, pp. 128–138, 1980.
31. M. Kuntz and K. Lampka, "Probabilistic meothds in state space analysis," in *Lecture Notes in Computer Science 2925/2004: Validation of Stochastic Systems*, pp. 339–383, 2004.
32. G. Holzmann, *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.
33. G. Holzmann, "An analysis of bitstate hashing," in *Proceedings of IFIP/PSTV95: Conference on Protocol Specification, Testing and Verification*, Warsaw, Poland: Chapman & Hall, June 1995.
34. B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, July 1970.
35. P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *Lecture Notes in Computer Science 697*, pp. 59–70, Springer Verlag, 1993.
36. U. Stern and D. Dill, "Improved probabilistic verification by hash compaction," in *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.
37. W. J. Knottenbelt, P. G. Harrison, M. S. Mestern, and P. S. Kritzinger, "A probabilistic dynamic technique for the distributed generation of very large state spaces," *Performance Evaluation*, vol. 39, pp. 127–148, February 2000.
38. M. Raynal, *Distributed Algorithms and Protocols.* John Wiley and Sons, 1988.
39. E. Dijkstra, W. Feijen, and A. Gasteren, "Derivation of a termination detection algorithm for distributed computations.," *Information Processing letters*, vol. 16, pp. 217–219, June 1983.
40. W. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.
41. R. Weiss, "A theoretical overview of Krylov subspace methods," *Applied Numerical Mathematics*, vol. 19, pp. 207–233, 1995. Special Issue on Iterative Methods for Linear Equations.
42. P. Sonneveld, "CGS, a fast Lanczos-type solver for nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, pp. 36–52, January 1989.
43. H. Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, pp. 631–644, March 1992.
44. R. Freund, "A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems," *SIAM Journal on Scientific Computing*, vol. 14, pp. 470–482, March 1993.
45. D. D. Deavours and W. H. Sanders, "An efficient disk-based tool for solving very large Markov models," in *TOOLS 1997, Computer Performance Evaluation: Modelling Techniques and Tools*, vol. 1245 of *Lecture Notes in Computer Science*, (St. Malo), pp. 58–71, Springer-Verlag, June 1997.

46. D. Deavours and W. Sanders, “An efficient disk-based tool for solving large Markov models,” *Performance Evaluation*, vol. 33, pp. 67–84, June 1998.
47. M. Kwiatkowska and R. Mehmood, “Out-of-core solution of large linear systems of equations arising from stochastic modelling,” in *Proceedings of Process Algebra and Performance Modelling (PAPM’02)*, (Copenhagen), pp. 135–151, July 25th–26th 2002.
48. R. Mehmood, “Serial disk-based analysis of large stochastic models,” in *Lecture Notes in Computer Science 2925/2004: Validation of Stochastic Systems*, pp. 230–255, 2004.
49. W. J. Knottenbelt and P. G. Harrison, “Distributed disk-based solution techniques for large Markov models,” in *NSMC’99, Proceedings of the 3rd Intl. Conference on the Numerical Solution of Markov Chains*, (Zaragoza), pp. 58–75, September 1999.
50. A. Bell and B. Haverkort, “Serial and parallel out-of-core solution of linear systems arising from Generalised Stochastic Petri Nets,” in *Proc. High Performance Computing Symposium (HPC 2001)*, pp. 242–247, 2001.
51. U. Catalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 673–693, July 1999.
52. G. Karypis and V. Kumar, *hMETIS: A Hypergraph Partitioning Package, Version 1.5.3.* University of Minnesota, November 1998.
53. A. Trifunovic and W. Knottenbelt, “Parkway 2.0: A parallel multilevel hypergraph partitioning tool,” in *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS’04)*, (Antalya, Turkey), October 27th–29th 2004.
54. P. G. Harrison, “Laplace transform inversion and passage-time distributions in Markov processes,” *Journal of Applied Probability*, vol. 27, pp. 74–87, March 1990.
55. J. Abate and W. Whitt, “The Fourier-series method for inverting transforms of probability distributions,” *Queueing Systems*, vol. 10, no. 1, pp. 5–88, 1992.
56. P. G. Harrison and W. Knottenbelt, “Passage time distributions in large Markov chains,” in *Proceedings of ACM SIGMETRICS 2002*, (Marina Del Rey, California), pp. 77–85, June 2002.
57. J. Abate and W. Whitt, “Numerical inversion of Laplace transforms of probability distributions,” *ORSA Journal on Computing*, vol. 7, no. 1, pp. 36–43, 1995.
58. J. Abate, G. L. Choudhury, and W. Whitt, “On the Laguerre method for numerically inverting Laplace transforms,” *INFORMS Journal on Computing*, vol. 8, no. 4, pp. 413–427, 1996.
59. P. G. Harrison and W. J. Knottenbelt, “Passage-time distributions in large Markov chains,” in *Proceedings of ACM SIGMETRICS 2002* (M. Martonosi and E. A. de Souza e Silva, eds.), pp. 77–85, Marina Del Rey, USA, June 2002.
60. J. T. Bradley, N. J. Dingle, W. J. Knottenbelt, and H. J. Wilson, “Hypergraph-based parallel computation of passage time densities in large semi-Markov models,” *Journal of Linear Algebra and Applications*, vol. 386, pp. 311–334, July 2004.
61. N. Dingle, *Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models.* PhD thesis, Imperial College London, October 2004.
62. W. Knottenbelt, “Generalised Markovian analysis of timed transition systems,” Master’s thesis, University of Cape Town, Cape Town, South Africa, July 1996.
63. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Cambridge, Massachussetts: MIT Press, 1994.

64. C. Woodside and Y. Li, “Performance Petri net analysis of communication protocol software by delay-equivalent aggregation,” in *Proceedings of the 4th International Workshop on Petri nets and Performance Models (PNPM’91)*, (Melbourne, Australia), pp. 64–73, IEEE Computer Society Press, 2–5 December 1991.

65. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Cambridge, Massachussetts: MIT Press, 1994.

66. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing*. Cambridge, Massachussetts: MIT Press, 1994.

67. J. T. Bradley, N. J. Dingle, W. J. Knottenbelt, and P. G. Harrison, “Performance queries on semi-Markov stochastic Petri nets with an extended Continuous Stochastic Logic,” in *PNPM’03, Proceedings of Petri Nets and Performance Models* (G. Ciardo and W. Sanders, eds.), (University of Illinois at Urbana-Champaign), pp. 62–71, IEEE Computer Society, September 2003.

68. S. Gilmore, J. Hillston, and G. Clark, “Specifying performance measures for PEPA,” in *Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems*, vol. 1601 of *Lecture Notes in Computer Science*, (Bamberg), pp. 211–227, Springer-Verlag, 1999.

69. J. Julvez, E. Jimenez, L. Recalde, and M. Silva, “On observability in timed continuous Petri net systems,” in *QEST’04, Proceedings of 1st International Conference on the Quantitative Evaluation of Systems*, (Enschede), pp. 60–69, IEEE Computer Society Press, September 2004.

70. J. Hillston, “Fluid flow approximation of PEPA models,” in *QEST’05, Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems*, (Torino), pp. 33–42, IEEE Computer Society Press, September 2005.

71. T. G. Kurtz, “Solutions of ordinary differential equations as limits of pure jump Markov processes,” *Journal of Applied Probability*, vol. 7, pp. 49–58, April 1970.