Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 908–920
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models

Nicholas J. Dingle,* Peter G. Harrison, and William J. Knottenbelt[1]

Department of Computing, Imperial College London, 180 Queens Gate, London SW7 2AZ, UK

Received 17 October 2002; revised 19 March 2004

Abstract

Fast response times and the satisfaction of response time quantile targets are important performance criteria for almost all transaction processing and computer-communication systems. We present a distributed uniformization-based technique for obtaining response time densities from very large unstructured Markov models. Our method utilizes hypergraph partitioning to minimize inter-processor communication while maintaining a good load balance. The resulting algorithm scales well on a distributed-memory parallel computer and, unusually for a problem of this nature, also produces near-linear speed-ups on a network of commodity PCs linked by 100 Mbps ethernet. We demonstrate our approach by calculating passage time densities in a 1.6 million state Markov chain derived from a Generalized Stochastic Petri net model and a 10.8 million state Markov chain derived from a closed tree-like queueing network. We compare the accuracy of our results with simulation and known analytical solutions and contrast the run-time performance of our technique with an approach based on numerical Laplace transform inversion.

Keywords: Response time densities and quantiles; Markov models; Hypergraph partitioning

1. Introduction

Stochastic performance models provide a formal way of capturing and analysing the dynamic behaviour of computer and communication systems. These models can be specified using several high-level formalisms including stochastic Petri nets [3], queueing networks [26] and stochastic process algebras [25]. With the exception of the special case of product-form queueing networks, performance statistics for these models are usually derived by generating and then solving a Markov chain corresponding to the model's behaviour at the state transition level. From the chain's equilibrium (steady state) probability distribution, standard resource-based performance measures (such as mean buffer occupancy, system availability and throughput) and *expected* values of various sojourn times can be obtained. There is a large body of previous work on the efficient calculation of steady-state probabilities in large Markov chains, including parallel [4,8,35] and disk-based [15,36,40] implementations, as well as those which employ implicit state-space representation techniques [12,16,24,41].

The focus of the present study, however, is on the harder problem of calculating full response time densities in structurally unrestricted Markov models. This has important practical applications since response time quantiles are often specified as quality of service metrics in Service Level Agreement contracts and industry standard benchmarks such as TPC-C [14]. 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 [20]. However, with the advent of high-performance parallel computing and the widespread availability of PC clusters, direct numerical analysis on Markov chains has now become a practical proposition.

There are two main methods for computing first passage time (and hence response time) densities in

*Corresponding author. Tel.: +44-20-7594-8385.
E-mail addresses: njd200@imperial.ac.uk (N.J. Dingle), pgh@doc.ic.ac.uk (P.G. Harrison), wjk@doc.ic.ac.uk (W.J. Knottenbelt).
[1] Also for correspondence

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.03.017

Markov chains: those based on Laplace transforms and their inversion [1,21] and those based on uniformization [42–44]. The former has wider application to semi-Markov processes but is less efficient than uniformization when restricted to Markov chains.

Our contribution is a parallel uniformization-based algorithm and tool for computing passage time densities in Markov chains with very large state spaces (more than 10^7 states). By using the state-of-the-art hypergraph partitioning techniques presented in [10] we achieve a remarkably scalable algorithm that yields excellent performance on a distributed-memory parallel computer and that effectively utilizes the compute power and RAM provided by a network of workstations. To the best of our knowledge, this is the first application of hypergraph partitioning in the domain of performance analysis. Further, we are not aware of any other distributed uniformization-based tools for computing response time densities, and our implementation improves substantially on both the solution time and capacity of contemporary distributed response time density analysers based on numerical Laplace transform inversion [21]. The uniformization-based approach of Miner [43] uses the implicit state-space representation technique of matrix diagrams to analyse systems of a similar size to those presented here for response time densities. This technique is not distributed, however, and like many approaches employing implicit techniques suffers from significant run-time performance overheads.

The rest of this paper is organized as follows. Section 2 summarizes the uniformization method and how it may be used to derive response time densities for arbitrary Markov chains. Section 3 compares and contrasts two data partitioning schemes for efficient parallel sparse matrix–vector multiplication: traditional graph-based partitioning and the recent hypergraph partitioning. Section 4 details the implementation of our parallel algorithm, which uses uniformization and hypergraph partitioning to produce response time density graphs for Markov chains derived from a high-level modelling formalism. Section 5 presents numerical results for two models, viz. a 1.6 million state Generalized Stochastic Petri net model and a 10.8 million state queueing network model, showing speed-up curves and validating against simulation and an exact analytical model, respectively. Section 6 concludes and discusses future work.

2. Response time densities via uniformization

2.1. First passage times

Consider a finite, irreducible, continuous-time Markov Chain (CTMC) with n states $\{1, 2, \ldots, n\}$ and 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 $q_{ii} = -\sum_{i \neq j} q_{ij}$. All state holding-times in a CTMC are exponentially distributed random variables. If $X(t)$ denotes the state of the CTMC at time $t \geqslant 0$, then the first passage time from a source state i into a non-empty set (denoted by a vector) of target states $\vec{j}$ is $(\forall t \geqslant 0)$

$$T_{i\vec{j}}(t) = \inf\{u > 0 : X(t+u) \in \vec{j} \mid X(t) = i\}.$$

For a stationary, time-homogeneous CTMC, $T_{i\vec{j}}(t)$ is independent of t, so

$$T_{i\vec{j}} = \inf\{u > 0 : X(u) \in \vec{j} \mid X(0) = i\}.$$

$T_{i\vec{j}}$ is a random variable with an associated probability density function $f_{i\vec{j}}(t)$ such that

$$\Pr(a < T_{i\vec{j}} \leqslant b) = \int_a^b f_{i\vec{j}}(t)\,\mathrm{d}t \quad (0 \leqslant a < b).$$

Our aim is to determine $f_{i\vec{j}}(t)$. In effect, this involves convolving exponentially distributed state holding times over all possible paths (including cycles) from state i into any of the states in the set $\vec{j}$. As we show in the next section, the problem can also be readily extended to multiple initial states by weighting the first passage time densities for each initial state.

2.2. Uniformization

Uniformization has classically been used to conduct transient analysis of finite-state, continuous-time Markov chains, see for example [18,46]. It involves the transformation of the 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 (DTMC), after normalization of the rows, together with an associated Poisson process of rate q. The one-step transition probability matrix $\mathbf{P}$ which characterizes the one-step behaviour of the DTMC is derived from the generator matrix $\mathbf{Q}$ of the CTMC as

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

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. The number of transitions in the DTMC that occur in a given time interval is given by a Poisson process with rate q.

Uniformization can also be employed for the calculation of response time densities in Markov chains as described in [42,44]. We add an extra, absorbing state to our uniformized chain, which is the sole successor state for all target states. This ensures we only calculate the first passage time density and need not worry about the case of successive visits to a target state. We denote by $\mathbf{P}'$ the one-step transition probability matrix of the modified, uniformized chain.

The calculation of the first passage time density between two states has two main components. The first considers the time to complete n hops ($n = 1, 2, 3, \ldots$). Recall that in the uniformized 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 uniformization, 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 the convolution of n exponentials all with rate q, which yields an n-stage Erlang density with rate 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 uniformized 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. (1)) drops below some threshold value. After this point, further terms are deemed to add nothing further to the response 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 uniformized 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 ($1 \leqslant n \leqslant m$). This can be generalized to allow for multiple target states in a straightforward manner, by providing a probability distribution across this set of states (such as the renormalized steady-state distribution calculated below in Eq. (3)).

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

$$f_{\vec{i}\vec{j}}(t) = \sum_{n=1}^{\infty} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right) \simeq \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in \vec{j}} \pi_k^{(n)} \right), \quad (1)$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P}' \quad \text{for } n \geqslant 0 \quad (2)$$

with

$$\pi_k^{(0)} = \begin{cases} 0 & \text{for } k \notin \vec{i}, \\ \pi_k / \sum_{j \in \vec{i}} \pi_j & \text{for } k \in \vec{i}. \end{cases} \quad (3)$$

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

$$\frac{||\boldsymbol{\pi}^{(n)} - \boldsymbol{\pi}^{(n-1)}||_\infty}{||\boldsymbol{\pi}^{(n)}||_\infty} < \varepsilon \quad (4)$$

is met, for given tolerance ε, the steady-state probabilities of $\mathbf{P}'$ are considered to have been obtained with sufficient accuracy and no further multiplications with $\mathbf{P}'$ are performed. Here, $||\mathbf{x}||_\infty$ is the infinity-norm given by $||\mathbf{x}||_\infty = \max_i |x_i|$.

3. Partitioning sparse matrices for parallel processing

The key opportunity for parallelism in the uniformization algorithm is the sparse matrix–vector product $\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \mathbf{P}'$, or equivalently $\boldsymbol{\pi}^{(n+1)\mathrm{T}} = \mathbf{P}'^{T} \boldsymbol{\pi}^{(n)\mathrm{T}}$, where the superscript T denotes the transpose operator. In the following, we let $\mathbf{A} = \mathbf{P}'^{\mathrm{T}}$, $\mathbf{x} = \boldsymbol{\pi}^{(n)\mathrm{T}}$ and consider sparse matrix–vector products of the form $\mathbf{Ax}$.

To perform this operation efficiently it is necessary to map the non-zero elements of $\mathbf{A}$ onto processors such that the computational load is balanced and communication between processors is minimized. One option proposed in [45] is to permute the rows and columns of the matrix randomly and then perform a 2D checkerboard partitioning [39]. For an $n \times n$ sparse matrix partitioned over p processors, this scheme achieves excellent load balance and an asymptotic worst-case communication overhead, per iteration, of $2\sqrt{p}(\sqrt{p}-1)$ messages of length $n/\sqrt{p}$, giving a total communication volume of $2n(\sqrt{p}-1)$. The alternative 2D checkerboard algorithm presented in [23] has worst-case communication requirements of $2p(\sqrt{p}-1)$ messages of length n/p, yielding the same total communication volume. The corresponding worst-case communication overhead for a random row-striped partitioning is $p(p-1)$ messages of length n/p, giving a total communication volume of $n(p-1)$.

The disadvantage of these approaches is that they are not scalable because their communication volume exceeds $O(n)$ while, in the context of Markov modelling, the computational cost is usually $O(n)$. This is because, typically, the number of non-zero elements in each row of the matrix (corresponding to the number of transitions out of the row-state) does not increase significantly with n.

Instead of randomly permuting the row and column indices, an alternative approach is to apply graph-based partitioning techniques in a row-striped decomposition to minimize inter-processor communication whilst maintaining a uniform balance of non-zero elements. In the following, we consider traditional graph-based and recent hypergraph techniques.

3.1. Graph partitioning

In a row-striped decomposition, the $n \times n$ sparse matrix $\mathbf{A}$ can be represented as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each row i $(1 \leqslant i \leqslant n)$ in the matrix corresponds to vertex $v_i \in \mathcal{V}$ in the graph. The corresponding weight w_i of vertex v_i is the total number of non-zeros in row i. For the edge-set $\mathcal{E}$, edge e_{ij} connects vertices v_i and v_j with weight 1 if either one of $a_{ij} > 0$ or $a_{ji} > 0$, and with weight 2 if both $a_{ij} > 0$ and $a_{ji} > 0$ [10].

The task of allocating the n rows of matrix $\mathbf{A}$ to p processors is well known to be equivalent to a p-way partitioning of the corresponding graph [31]. The quality of such a decomposition is judged with respect to two metrics: edge cut and balance. An edge is *cut* if the vertices which it connects are assigned to two different processors—so that the total number of edges cut is an approximation for the amount of interprocessor communication. A decomposition is said to be *balanced* if the sum of the weights of the vertices in each partition does not differ from the average of these weight sums by more than a specified amount. An optimal decomposition is one which minimizes edge cut while satisfying the balance constraint.

The problem of finding the optimal decomposition for a given graph is NP-complete. However, there exist a number of tools which implement heuristic algorithms to calculate good sub-optimal decompositions, for example Chaco [22] and METIS [27,29]. A parallel implementation of METIS called ParMETIS [32,33] is also available. ParMETIS is particularly attractive for very large matrices as an arbitrary number of processors may be used to calculate the p-way partition, and per-processor memory use is inversely proportional to the number of processors.

Consider the problem of producing a 4-way row-wise decomposition of the matrix shown at the top of Fig. 1. The matrix on the bottom left of Fig. 1 shows the matrix and vector partitioning produced by the graph partitioning tool Chaco. Note how the effect of the

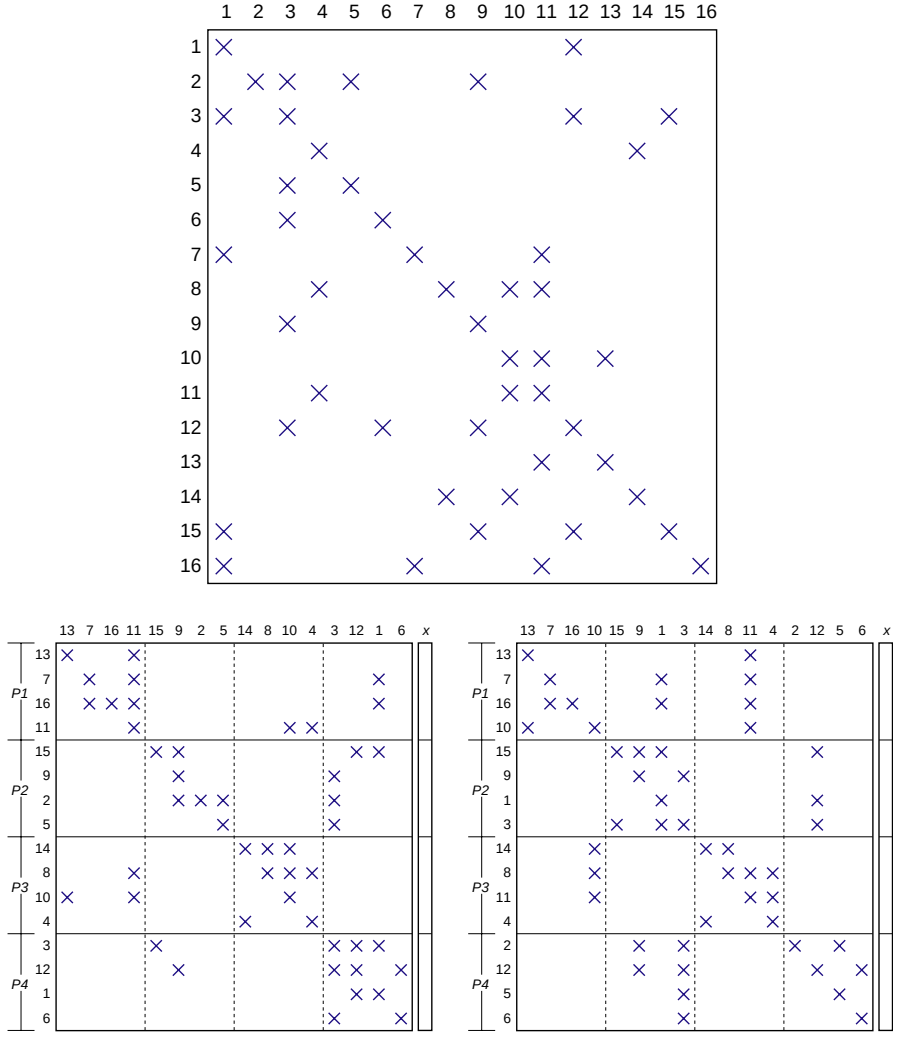

Fig. 1. A 16×16 non-symmetric sparse matrix $\mathbf{A}$ (top), with corresponding 4-way graph partition (left bottom) and 4-way hypergraph partition (right bottom) and corresponding partitions of the vector $\mathbf{x}$.

decomposition has been to minimize the number of non-zeros that occur in off-diagonal blocks (just 14 off-diagonal elements as opposed to 27 in the original matrix). However, while the edge cut is 14, the number of vector elements that must be sent between processors (i.e. the real communication cost) is just 10. This is because off-diagonal non-zeros which are in the same column and on the same processor are all multiplied by the same remote vector element, a factor which is not accounted for by graph-based partitioning strategies.

3.2. Hypergraph partitioning

Hypergraph partitioning is an extension of graph partitioning. Its primary application has been in VLSI circuit design where the objective is to cluster pins of devices such that interconnect is minimized.

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 $\mathcal{V}$ [5,6]. In the context of a row-wise decomposition of a sparse matrix as described in [10], matrix row i $(1 \leqslant i \leqslant n)$ is represented by a vertex $v_i \in \mathcal{V}$ while column j $(1 \leqslant j \leqslant n)$ 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 and only if $a_{ij} \neq 0$. Weights are assigned to vertices in the same manner as to the vertices of a graph i.e. the weight of vertex i is given by the number of non-zero elements in row i. The weight of all nets is one, with a net's contribution to the hyperedge cut being defined as one less than the number of different partitions (in the row-wise decomposition) spanned by that net. The overall objective of a hypergraph sparse matrix partitioning is to minimize the hyperedge cut while maintaining a balance criterion. This corresponds to minimizing the total communication volume whilst maintaining computational load balance when performing sparse matrix–vector multiplication in parallel. In this context, we apply a hypergraph partition to the corresponding matrix by symmetrically permuting the rows and columns of the matrix such that all rows corresponding to vertices in a partition are assigned to one processor.

The matrix on the bottom right of Fig. 1 shows the result of applying hypergraph-partitioning to the matrix at the top. Although the number of off-diagonal non-zeros has increased from 14 to 18 compared with the graph decomposition, the number of vector elements which must be transmitted between processors (the communication cost) has dropped from 10 to 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 hyperedge cut exactly quantifies the communication cost, unlike the edge cut in graph partitioning. This is a general property and one of the key advantages of using hypergraphs.

Like graph partitioning, optimal hypergraph partitioning is NP-complete. However, there exist a small number of hypergraph partitioning tools which implement fast heuristic algorithms, for example PaToH [10,11] and hMETIS [30,28]. These are all written to run on a single processor so their capacity is limited to models with a few million states. We have identified a parallel hypergraph partitioner capable of functioning on very large models as a future research area. We note that, for very large models, a parallel graph partitioner still yields a great reduction in communication costs over other methods (see Section 5.2 for an example).

4. Parallel algorithm and tool implementation

The process of calculating a response time density begins with a high-level model, which we specify in an enhanced form of the DNAmaca Markov Chain Analyser interface language [34,35]. This language supports the specification of queueing networks, stochastic Petri nets, stochastic process algebras and other models that can be mapped onto Markov chains. Next, a probabilistic, hash-based state generator [37] uses the high-level model description to produce the generator matrix $\mathbf{Q}$ of the model's underlying Markov chain as well as a list of the initial and target states. We also calculate $\mathbf{P}$ from $\mathbf{Q}$ (as shown in Section 2.2). Normalized weights for the initial states are then determined from Eq. (3), which requires us to solve $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$. This is readily done using any of a variety of steady-state solution techniques (e.g. [15,36]). $\mathbf{P}'^{\mathrm{T}}$ is constructed from $\mathbf{P}$ by transposing and by adding the extra, terminal state that becomes the sole successor state of all target states. Having been converted into an appropriate input format, $\mathbf{P}'^{\mathrm{T}}$ is then partitioned using a hypergraph or graph-based partitioning tool.

The analysis pipeline is completed by our distributed response time density calculator, which is implemented in C++ using the Message Passing Interface (MPI) [19] standard. This means that it is portable to a wide variety of parallel computers and workstation clusters. Initially, each processor tabulates the Erlang terms for each t-point required (cf. Eq. (1)). Computation of these terms terminates when they fall below a specified threshold value. In fact, this is safe to use as a truncation condition for the entire passage time density expression because the Erlang term is multiplied by a summation which is a probability. The terminating condition also determines the maximum number of hops m used to calculate the inner summation in Eq. (1), which is independent of t.

Each processor reads in the rows of the matrix $\mathbf{P}'^{\mathrm{T}}$ that correspond to its allocated partition into two types

of sparse matrix data structure and also computes the corresponding elements of the vector $\boldsymbol{\pi}^{(0)}$. *Local* non-zero elements (i.e. those elements in the diagonal matrix blocks that will be multiplied with vector elements stored locally) are stored in compressed sparse row (CSR) format (see [47, p. 153]). *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. That is to say, each non-zero is stored in the form `<rowIndex> <columnIndex> <nonZeroValue>`. Each processor then determines the vector elements which will need to be received from and sent to every other processor on each iteration, adjusting the column 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 very efficient.

The vector $\boldsymbol{\pi}^{(n)}$ is then calculated for $n = 1, 2, 3, \ldots, m$ by repeated sparse matrix–vector multiplications of form $\boldsymbol{\pi}^{(n+1)\mathrm{T}} = \mathbf{P}'^{\mathrm{T}}\boldsymbol{\pi}^{(n)\mathrm{T}}$. Actually, fewer than m multiplications may take place since a test for steady-state convergence is made after every iteration (cf. Eq. (4)); if the convergence criterion is satisfied, the matrix–vector multiplication is not performed and we set $\boldsymbol{\pi}^{(n+1)\mathrm{T}} = \boldsymbol{\pi}^{(n)\mathrm{T}}$ in subsequent iterations. The check for convergence is performed on each processor individually and the results broadcast to every other processor. Only if the calculations on all processors have converged do we stop performing the multiplications. The broadcasting of convergence results is, therefore, a synchronisation point in the algorithm.

For each matrix–vector multiplication, 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 matrix–vector product cumulatively.

From the resulting local matrix–vector products each processor calculates and stores its contribution to the sum $\sum_{k\in\vec{j}}\pi_k^{(n)}$. After m iterations have completed, these sums are accumulated onto an arbitrary master processor where they are multiplied with the tabulated Erlang terms for each t-point required for the passage time density. The resulting points are written to a disk file and are displayed using the GNUplot graph plotting utility.

5. Numerical results

This section presents numerical results that demonstrate the applicability, accuracy, scalability and capacity of our technique. First, we compute a first passage time density in a Petri net model of a manufacturing system. We consider the scalability of our algorithm on two different parallel architectures and validate the density produced against a simulation. Next, we compute a cycle time density in a queueing network model with a very large underlying Markov chain. We illustrate the effect of hypergraph partitioning and compare the results with an analytical solution. Finally, we compare the time taken to partition and solve a model using hypergraph partitioning with the corresponding time to perform the calculations using a row-striped (linear) partition.

5.1. The FMS generalized stochastic Petri net model

Fig. 2 shows a 22-place Generalized Stochastic Petri net (GSPN) [2] model of a flexible manufacturing system. Interested readers are directed to [3] as a good introduction to GSPNs, while a full description of this model, which we will refer to as the FMS model, can be found in [13]. For our purposes, it suffices to note that the model describes an assembly line with three types of machines ($M1$, $M2$ and $M3$) which assemble four types of parts ($P1$, $P2$, $P3$ and $P12$). Initially, there are k

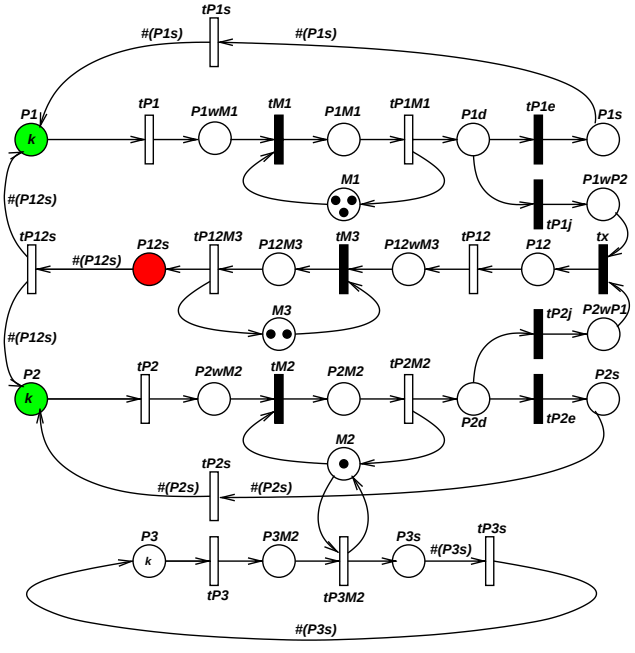

Fig. 2. The flexible manufacturing system (FMS) GSPN [13].

unprocessed parts of each type $P1$, $P2$ and $P3$ in the system. There are no parts of type $P12$ at start-up since these are assembled from processed parts of type $P1$ and $P2$ by the machines of type $M3$. When parts of any type are finished, they are stored for shipping on places $P1s$, $P2s$, $P3s$ and $P12s$.

For $k = 7$, the GSPN's underlying Markov chain has 1 639 440 states and 13 552 968 non-zero off-diagonal entries in its generator matrix $\mathbf{Q}$. For this model, we calculate the density of the time taken to produce a finished part of type $P12$ starting from any state in which there are 7 unprocessed parts of type $P1$ and 7 unprocessed parts of type $P2$. That is, the source markings (of which there are 36, corresponding to the possible submarkings of $M3$) are those where $M(P1) = M(P2) = 7$ and the target markings (of which there are 429 624) are those where $M(P12s) = 1$. We weight the density from each source state according to the relative probability that the passage originates in that state (cf. Eq. (3)).

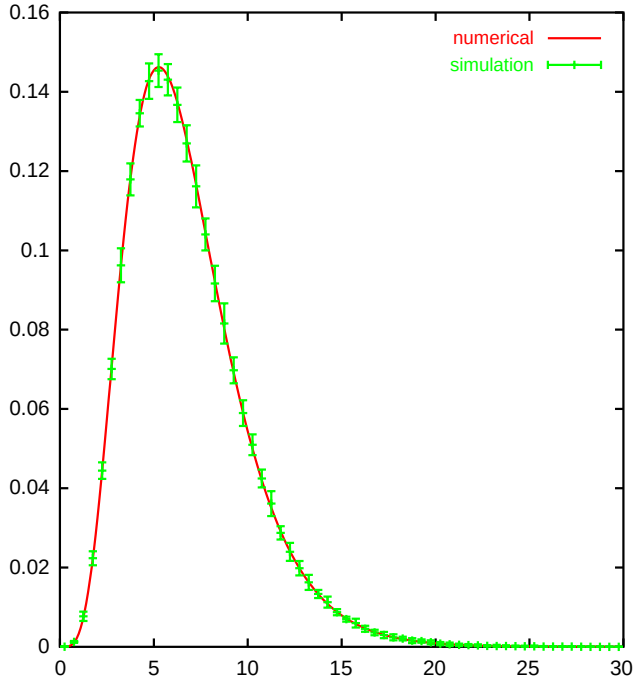

Fig. 3. Numerical and simulated (with 95% confidence intervals) passage time densities for the time taken to produce a finished part of type $P12$ starting from states in which there are $k = 7$ unprocessed parts of types $P1$ and $P2$.

After modification of the state graph to allow for transitions from target states to a new terminal state, the uniformized matrix $\mathbf{P}'$ has 11 001 408 non-zero entries. The hypergraph tool PaToH is then used to partition the rows of the transposed matrix $\mathbf{P}'^{\mathrm{T}}$ as input to our parallel algorithm. Fig. 3 shows the resulting numerically calculated passage time density, which is validated against the combined results from 10 simulations (each of which consisted of 1 billion transition firings) plotted with 95% confidence bounds. There is excellent agreement between the numerical and simulated passage time densities.

Table 1 shows the performance of our algorithm on two architectures: a Fujitsu AP3000 distributed-memory parallel computer running Solaris and a Linux-based PC workstation cluster. The AP3000 is based on a grid of 60 processing nodes, each of which has a UltraSPARC 300 MHz processor and 256 MB RAM. These nodes are interconnected by a 2D wraparound mesh network that uses wormhole routing and that has a peak throughput of 520 Mbps (megabits per second). The PC cluster is a vanilla network of workstations, consisting of 32 Athlon 1.4 GHz PCs each with 512 MB RAM linked together by a 100 Mbps switched Ethernet network. Distributed run-time is measured as the maximum processor run time from the start of the first uniformization iteration. The speedup for p processors, denoted by S_p, is given by the run time of the sequential solution ($p = 1$) divided by the run time with p processors. Efficiency for p processors, denoted by E_p, is defined as $E_p = S_p/p$. In every case, the sparse matrix was partitioned using PaToH on an Intel Pentium 4 2.6 GHz machine with 1GB of RAM using the following partitioning options:

`OCM RA = 10, MT = 12, WI = 1, FI = 0.05.`

That is, the hypergraph is derived from a sparse matrix and should be partitioned using the Boundary FM

Table 1
Run time, speedup (S_p), efficiency (E_p) and per-iteration communication overhead for p-processor passage time density calculation in the FMS model with $k = 7$

	AP3000			PC cluster			Comm. per iteration	
p	Time (s)	S_p	E_p	Time (s)	S_p	E_p	Messages	Vol (MB)
1	1243.3	1.00	1.000	325.0	1.00	1.000	0	0
2	630.5	1.97	0.986	258.7	1.26	0.628	2	1.5
4	328.2	3.78	0.947	197.1	1.65	0.412	12	3.2
8	182.3	6.82	0.853	143.0	2.27	0.284	51	5.3
16	99.7	12.47	0.779	114.6	2.84	0.178	207	7.3
32	58.6	21.22	0.663	71.7	4.53	0.142	663	9.6

Results are presented for an AP3000 distributed-memory parallel computer and a PC cluster.

refinement algorithm [17] with Krishnamurthy's multi-level gain [38], the absorption clustering using pins coarsening algorithm [11] and a permitted imbalance between final partitions of 5%.

Corresponding graphs of the run time, speedup and efficiency on each architecture are presented in Figs. 4 and 5. The speedups and efficiencies achieved on the AP3000 are excellent. Solution time on a single AP3000 node is 20 min 43 s whereas on 32 processors it takes just 58.6 s (i.e. 21.22 times faster, corresponding to an efficiency of 66.3%).

With processors that are about 4 times faster and a communication network that is about 6 times slower than the AP3000, and without exclusive access to either processors or the interconnection network, we cannot expect such good results on the (shared departmental) PC cluster. However, unusually for problems of this type, reasonable speedups are still achieved, requiring 5 min 25 s on a single PC and 1 min 12 s on 32 PCs (i.e. 4.53 times faster, corresponding to an efficiency of 14.2%). The speedup trend for the PC cluster is shallow but linear in trend, suggesting that speedup will continue to improve for an even larger number of processors. Adding extra workstations also boosts solution capacity through additional RAM. Note that the results presented for the PC cluster were gathered at times when the network and processors were most likely to be idle (e.g. late at night) and have been averaged over several runs to minimize the impact of any external interference.

Not only does our distributed algorithm exhibit scalability but it is also efficient in absolute terms—using a technique based on Laplace transform inversion to calculate the same passage time density requires 1566 s (26 min 6 s) on 32 PCs [21].

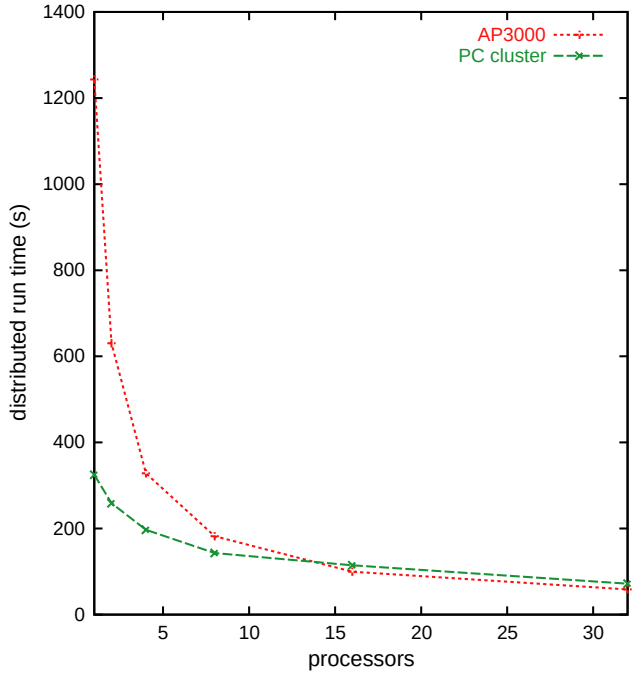

Fig. 4. Distributed run time for the FMS model with $k = 7$ on the AP3000 and a PC cluster.

5.2. A tree-like queueing network

The second example we consider is a cycle time in the closed tree-like queueing network of Fig. 6. This network has six servers with rates $\mu_1, \ldots, \mu_6$ and non-zero routing probabilities as shown. Thus the visitation rates $v_1, \ldots, v_6$ for servers 1–6 are, respectively, proportional to: $1, p_{12}, p_{13}, p_{14}, p_{12}, p_{14}$. For this example, we set $\{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6\} = \{3, 5, 4, 6, 2, 1\}$ and $\{p_{12}, p_{13}, p_{14}\} = \{0.2, 0.5, 0.3\}$.

Analytical results for the cycle time density in this type of overtake-free, tree-like queueing network with M servers and population n are known [20,21]. For interested readers, the corresponding algorithm is given in Appendix A. To compute the cycle time density in this network in terms of its underlying Markov Chain using

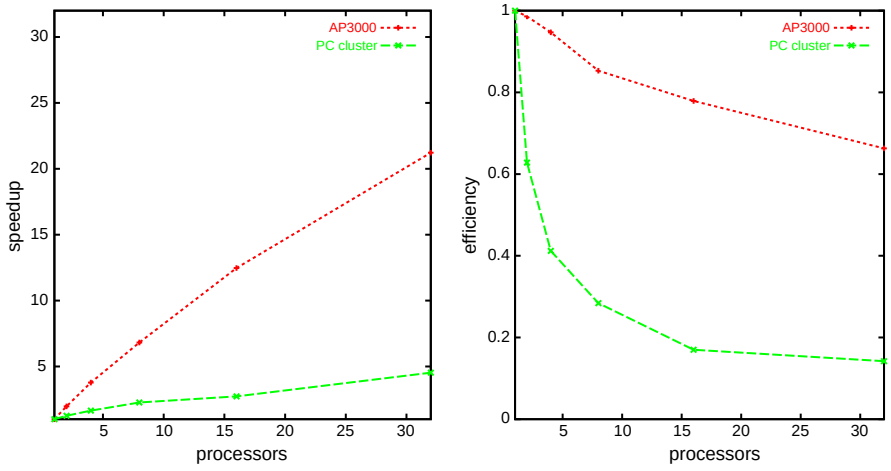

Fig. 5. Speedup (left) and efficiency (right) for the FMS model with $k = 7$ on the AP3000 and a PC cluster.

the uniformization technique described in this paper requires the state vector to be augmented by 3 extra components so that a "tagged" customer can be followed through the system. The extra components are: the queue containing the tagged customer l, the position of the tagged customer in that queue k (with $k \geqslant 0$), and the cycle sequence number c (an alternating bit, flipped whenever the tagged customer joins $q1$). For this augmented system with n customers, the underlying Markov chain has $12\binom{n+5}{6}$ states. Source states are those in which $l = 1$, $k = n_1 - 1$ and $c = 0$ while target states are those in which $l = 1$, $k = n_1 - 1$ and $c = 1$, where n_1 is the queue length of q_1.

For a small six customer system with 5544 states, Fig. 7 shows the resulting transposed $\mathbf{P}'$ matrix and associated hypergraph decomposition produced by hMETIS for a 4 processor decomposition. Statistics about the per-iteration communication associated with this decomposition are presented in Table 2. 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 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 Table 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).

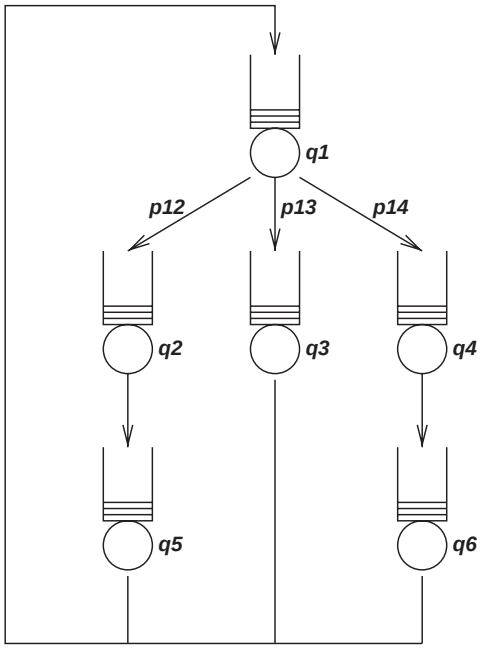

Fig. 6. A tree-like network and its routing probabilities.

Moving to a more sizeable model, the queueing network with 27 customers has an underlying Markov Chain with 10 874 304 states and 82 883 682 transitions. This model is too large to partition using a hypergraph partitioner on a single machine (even one with 2GB RAM), and there are currently no parallel hypergraph partitioning tools available. Consequently a lesser quality graph-based decomposition produced by the parallel graph partitioner ParMETIS (running on the PC cluster) was chosen. The options chosen were to use the parallel partitioning algorithm, a successive folding level of 300 [33] and weights on both vertices and edges. It must be noted that this decomposition still offers a great reduction in communication costs over other methods available: a 16-way partition has an average of 95.8% local non-zero elements allocated to each processor and a reused received non-zero element average of 30.4%. Table 3 shows the per-iteration communication overhead for randomized (i.e. random assignment of rows to partitions), linear (i.e. simple in-order allocation of rows to processors such that the number of non-zeros assigned to each processor is the same) and graph-based allocations. The graph-based method is clearly superior,

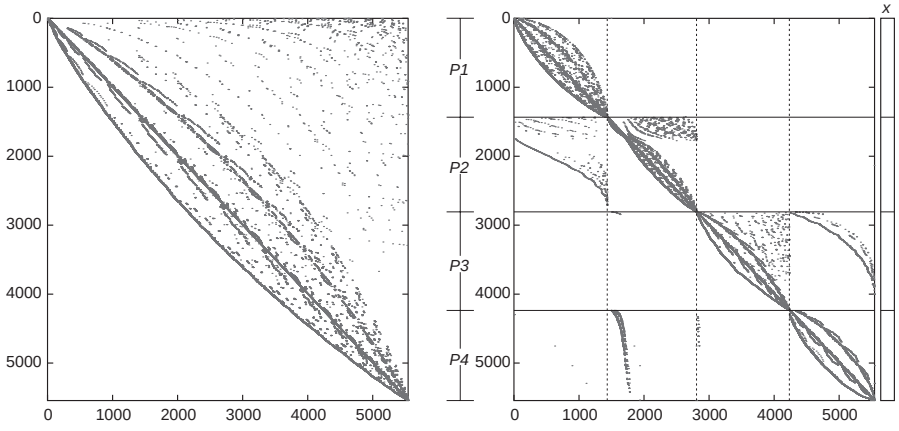

Fig. 7. Transposed $\mathbf{P}'$ matrix (left) and hypergraph-partitioned matrix (right) for the tree-like queueing network with 6 customers (5544 states).

Table 2
Communication overhead in the queueing network model with six customers (left) and interprocessor communication matrix (right) for each processor in a 4 processor decomposition

Processor	Non-zeros	Local (%)	Remote (%)	Reused (%)		1	2	3	4
1	7022	99.96	0.04	0	1	—	407	—	4
2	7304	91.41	8.59	34.93	2	3	—	16	181
3	6802	88.44	11.56	42.11	3	—	—	—	12
4	6967	89.01	10.99	74.28	4	—	1	439	—

Table 3
Per-iteration communication overhead for various partitioning methods for the queueing network model with 27 customers on 16 processors

Partitioning method	Communication overhead	
	Messages	Volume (MB)
Randomised	240	450.2
Linear	134	78.6
Graph-based	110	19.7

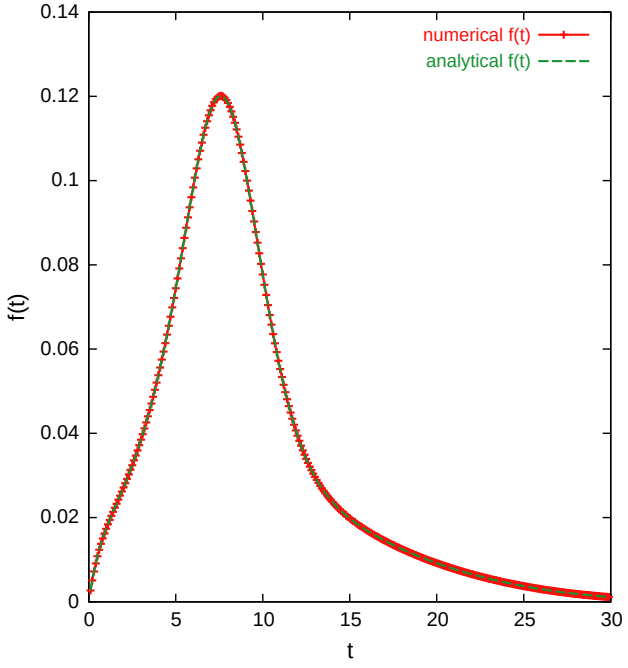

Fig. 8. Numerical and analytical cycle time densities for the tree-like queueing network of Fig. 6 with 27 customers (10 874 304 states).

both in terms of number of messages sent and (especially) communication volume.

Fig. 8 compares the numerical and analytical cycle time densities for the queueing network with 27 customers. Agreement is excellent and the results agree to an accuracy of 0.00001% over the time range plotted. The numerical density is computed in 968 s (16 min 8 s) for 875 iterations using 16 PCs. The memory used on each PC is just 84 MB. It was not possible to compute the density on a single PC (with 512 MB RAM) but the same computation on a dual-processor server machine (with 2 GB RAM) required 5580 s (93 min).

5.3. Evaluation

Partitioning a sparse matrix for parallel sparse matrix–vector multiplication using hypergraph partitioning aims to reduce the amount of data which much be exchanged at each step. A key consideration, however, is how much time is saved by doing this—in particular, is it quicker to simply naïvely partition the matrix by row and then do the multiplications at higher cost than it is to calculate a hypergraph partition and then use it in the multiplications?

Table 4 compares the partitioning time and multiplication time for hypergraph-partitioned and linear row-striped matrix–vector multiplication on the two different architectures described in Section 5.1 for the analysis of the FMS model. On both architectures we observe that the run-time for hypergraph-partitioned matrix–vector multiplication is lower than that of linear row-striped multiplication for all numbers of processors. It is also noticeable that hypergraph-partitioned multiplication scales far better than linear row-striped multiplication on the PC cluster. On the AP3000, where the network is faster and the processors slower, the difference is still observed but does not favour the hypergraph-partitioned scheme as much.

Note, however, that for large numbers of processors (typically, 8 or more) the time to perform the multiplication and the partitioning is higher for the hypergraph scheme than the row-striped scheme. We offer three observations regarding this: firstly, a current area of research is the development of a scalable parallel hypergraph partitioner and so we can expect the overhead of calculating the partition to reduce. Secondly, for a given model and set of target states, the hypergraph partition is reusable (so if we wish to calculate a response time from a different set of source states or over a different time range we do not need to recalculate the partition). Finally, in the example presented here several hundred successive matrix–vector multiplications are performed, but in other techniques for response time density calculation (e.g. [7]) many millions of such operations are performed, thus reducing

Table 4
Run-times for hypergraph and linear row-striped parallel sparse matrix–vector multiplication

p	Hypergraph partitioning (s)	PC time (s)	AP time (s)	Row-partitioning time (s)	PC time (s)	AP time (s)
1	N/A	325.0	1243.3	N/A	325.0	1243.3
2	66.96	258.7	630.5	6.07	635.3	817.4
4	197.12	197.1	328.2	5.61	569.4	484.9
8	266.39	143.0	182.3	5.65	388.3	283.0
16	323.29	114.6	99.7	5.92	362.9	163.0

the relative overhead of the hypergraph partitioning step.

6. Conclusion

We have developed a scalable, parallel, uniformization-based algorithm that computes passage time densities in very large Markov chains (over 10^7 states). The method has been validated using both simulation and exact analytical results, and found to be extremely accurate. This capability facilitates the detailed analysis of quality of service in non-trivial high-level models previously considered intractable. In view of the scalability achieved, it would be possible to extend the approach to even larger state spaces—perhaps by two orders of magnitude. This could be accomplished by employing a disk-based algorithm [15] together with the increased RAM and processing power that would be provided by more nodes.

Key to our scalability are the graph and hypergraph partitioning schemes employed. Our results suggest an important area for future research, viz. development of scalable algorithms for parallel hypergraph partitioning—for preliminary work in this area see [48]. Apart from the objective described above, this would find application in the computation of equilibrium state probabilities in very large Markov chains, as well as in other fields such as VLSI design.

Acknowledgments

The authors would like to thank Jeremy Bradley, Tony Field, Paul Kelly and David Thornley for their helpful comments and advice and the Imperial College Parallel Computing Centre for the use of the AP3000 distributed-memory parallel computer. We would also like to thank the anonymous referees for their constructive and considered suggestions.

Appendix A. Analytical cycle time calculation in the tree-like queueing network

If the servers in a closed tree-like queueing network in an overtake-free path $(1, 2, \ldots, m)$ $(m \leqslant M)$ have distinct service rates $\mu_1, \mu_2, \ldots, \mu_m$, the passage time density function, conditional on the choice of path, is

$$\frac{\prod_{i=1}^{m} \mu_i}{G(n-1)} \sum_{c=0}^{n-1} G_m(n-c-1) \times \sum_{j=1}^{m} \frac{e^{-\mu_j t}}{\prod_{1 \leqslant i \neq j \leqslant m} (\mu_i - \mu_j)} \sum_{i=0}^{c} \frac{(v_j t)^{c-i}}{(c-i)!} K^m(j,i),$$

where $K^m(j,l)$, $G_m(n-c-1)$ and $G(n-1)$ are normalizing constants that may be computed efficiently by Buzen's algorithm [9]. If we define the recursive function k, for real vector $\mathbf{y} = (y_1, \ldots, y_a)$ and integers a, b $(0 \leqslant a \leqslant M,\ 0 \leqslant b \leqslant N-1)$ by

$$k(\mathbf{y}, a, b) = k(\mathbf{y}, a-1, b) + y_a k(\mathbf{y}, a, b-1) \quad (a, b > 0),$$

$$k(\mathbf{y}, a, 0) = 1 \quad (a > 0),$$

$$k(\mathbf{y}, 0, b) = 0 \quad (b \geqslant 0),$$

then

$$G_m(l) = k(\mathbf{x}_m, M-m, l) \quad (0 \leqslant l \leqslant n-1),$$

$$G(n-1) = k(\mathbf{x}, M, n-1),$$

$$K^m(j,l) = k(\mathbf{w}_j, m-1, l),$$

with $x_i = v_i/\mu_i$, $\mathbf{x} = (x_1, \ldots, x_M), \mathbf{x}_m = (x_{m+1}, \ldots, x_M)$ and, for $1 \leqslant j \leqslant m$,

$$(\mathbf{w}_j)_k = \begin{cases} (v_k - v_j)/(\mu_k - \mu_j) & \text{if } 1 \leqslant k < j, \\ (v_{k+1} - v_j)/(\mu_{k+1} - \mu_j) & \text{if } j \leqslant k < m. \end{cases}$$

References

[1] J. Abate, W. Whitt, The Fourier-series method for inverting transforms of probability distributions, Queueing Systems 10 (1) (1992) 5–88.
[2] M. Ajmone-Marsan, G. Conte, G. Balbo, A class of generalised stochastic Petri nets for the performance evaluation of multiprocessor systems, ACM Trans. Comput. Systems 2 (1984) 93–122.
[3] F. Bause, P.S. Kritzinger, Stochastic Petri Net Theory, Verlag Vieweg, Wiesbaden, Germany, 1995.
[4] M. Benzi, M. Tuma, A parallel solver for large-scale Markov chains, Appl. Numer. Math. 41 (2002) 135–153.
[5] C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, 1973.
[6] C. Berge, Hypergraphs: Combinatorics of Finite Sets, North-Holland, Amsterdam, 1989.

[7] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, H.J. Wilson, Hypergraph-based parallel computation of passage time densities in large semi-Markov processes, in: Proceedings of the Fourth International Conference on the Numerical Solution of Markov Chains (NSMC '03), Urbana-Champaign, IL, September 2003, pp. 99–120.
[8] P. Buchholz, M. Fischer, P. Kemper, Distributed steady state analysis using Kronecker algebra, in: Proceedings of the Third International Conference on the Numerical Solution of Markov Chains (NSMC '99), Zaragoza, Spain, September 1999, pp. 76–95.
[9] J.P. Buzen, Computational algorithms for closed queueing networks with exponential servers, Comm. ACM 16 (1973) 527–531.
[10] U.V. Catalyürek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distributed Systems 10 (7) (1999) 673–693.
[11] U.V. Catalyürek, C. Aykanat, PaToH: a multilevel hypergraph partitioning tool, Technical Report BU-CE-9915, Version 3.0, Department of Computer Engineering, Bikent University, Ankara, Turkey, 1999.
[12] G. Ciardo, A.S. Miner, A data structure for the efficient Kronecker solution of GSPNs, in: Proceedings of the Eighth International Conference on Petri Nets and Performance Models (PNPM '99), Zaragoza, Spain, September 1999, pp. 22–31.
[13] G. Ciardo, K.S. Trivedi, A decomposition approach for stochastic reward net models, Performance Evaluation 18 (1) (1993) 37–59.
[14] Transaction Processing Performance Council, TPC benchmark C: standard specification revision 5.2, December 2003.
[15] D.D. Deavours, W.H. Sanders, An efficient disk-based tool for solving large Markov models, Performance Evaluation 33 (1) (1998) 67–84.
[16] D.D. Deavours, W.H. Sanders, "On-the-fly" solution techniques for stochastic Petri nets and extensions, IEEE Trans. Software Eng. 24 (10) (1998) 889–902.
[17] C.M. Fiduccia, R.M. Mattheyses, A linear time heuristic for improving network partitions, in: Proceedings of the 19th IEEE Design Automation Conference, Las Vegas, NV, 1982, pp. 175–181.
[18] W. Grassman, Means and variances of time averages in Markovian environments, Eur. J. Oper. Res. 31 (1) (1987) 132–139.
[19] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, Cambridge, MA, 1994.
[20] P.G. Harrison, Laplace transform inversion and passage-time distributions in Markov processes, J. Appl. Probab. 27 (1990) 74–87.
[21] P.G. Harrison, W.J. Knottenbelt, Passage time distributions in large Markov chains, in: Proceedings of ACM SIGMETRICS 2002, Marina Del Rey, CA, June 2002, pp. 77–85
[22] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of ACM/IEEE Supercomputing Conference, ACM/IEEE, New York, December, 1995.
[23] B. Hendrickson, R. Leland, S. Plimpton, An efficient parallel algorithm for matrix–vector multiplication, Internat. J. High Speed Comput. 7 (1) (1995) 73–88.
[24] H. Hermanns, J. Meyer-Kayser, M. Siegle, Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains, in: Proceedings of the Third International Conference on the Numerical Solution of Markov Chains (NSMC '99), Zaragoza, Spain, September 1999, pp. 188–207.
[25] J. Hillston, A compositional approach to performance modelling, Ph.D. Thesis, University of Edinburgh, 1994.
[26] Ng Chee Hock, Queueing Modelling Fundamentals, Wiley, New York, 1996.
[27] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, Technical Report #95-035, University of Minnesota, 1998.
[28] G. Karypis, V. Kumar, hMETIS: A Hypergraph Partitioning Package, Version 1.5.3, University of Minnesota, November 1998.
[29] G. Karypis, V. Kumar, METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0, University of Minnesota, September 1998.
[30] G. Karypis, V. Kumar, Multilevel k-way hypergraph parititioning, Technical Report #98-036, University of Minnesota, 1998.
[31] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distributed Comput. 48 (1) (1998) 96–129.
[32] G. Karypis, V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs, Technical Report #96-036, University of Minnesota, 1998.
[33] G. Karypis, K. Schloegel, V. Kumar, ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0, University of Minnesota, September 2002.
[34] W.J. Knottenbelt, Generalised Markovian analysis of timed transition systems, Master's Thesis, University of Cape Town, Cape Town, South Africa, July 1996.
[35] W.J. Knottenbelt, Parallel Performance Analysis of Large Markov Models, Ph.D. Thesis, Imperial College, London, United Kingdom, February 2000.
[36] W.J. Knottenbelt, P.G. Harrison, Distributed disk-based solution techniques for large Markov models, in: Proceedings of the Third International Conference on the Numerical Solution of Markov Chains (NSMC '99), Zaragoza, Spain, September 1999, pp. 58–75.
[37] W.J. Knottenbelt, P.G. Harrison, M.A. Mestern, P.S. Kritzinger, A probabilistic dynamic technique for the distributed generation of very large state spaces, Performance Evaluation 39 (1–4) (2000) 127–148.
[38] B. Krishnamurthy, An improved min-cut algorithm for partitioning VLSI networks, IEEE Trans. Comput. 33 (5) (1984) 438–446.
[39] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin/Cummings Publishing, Menlo Park, CA, 1994.
[40] M. Kwiatkowska, R. Mehmood, Out-of-core solutions of large linear systems of equations arising from stochastic modelling, in: Proceedings on Process Algebra and Performance Modelling (PAPM '02), Copenhagen, July 2002, pp. 135–151.
[41] M. Kwiatkowska, G. Norman, D. Parker, PRISM: Probabilistic symbolic model checker, in: Lecture Notes in Computer Science, Vol. 2324: Proceedings of the 12th International Conference on Modelling, Techniques and Tools (TOOLS 2002), London, UK, Springer, Berlin, 2002, pp. 200–204.
[42] B. Melamed, M. Yadin, Randomization procedures in the computation of cumulative-time distributions over discrete state Markov processes, Oper. Res. 32 (4) (1984) 926–944.
[43] A.S. Miner, Computing respone time distributions using stochastic Petri nets and matrix diagrams, in: Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM 2003), Urbana-Champaign, IL, September 2003, pp. 10–19.
[44] J.K. Muppala, K.S. Trivedi, Numerical transient analysis of finite Markovian queueing systems, in: U.N. Bhat, I.V. Basawa (Eds.), Queueing and Related Models, 1992, pp. 262–284.
[45] A.T. Ogielski, W. Aiello, Sparse matrix computations on parallel processor arrays, SIAM J. Scient. Comput. 14 (3) (1993) 519–530.
[46] A. Reibman, K.S. Trivedi, Numerical transient analysis of Markov models, Comput. Operat. Res. 15 (1) (1988) 19–36.
[47] W.J. Stewart, Introduction to the Numerical Solution of Markov Chains, Princeton University Press, Princeton, NJ, 1994.

[48] A. Trifunovic, W.J. Knottenbelt, Towards a parallel algorithm for multilevel k-way hypergraph partitioning, in: Proceedings of the Fifth Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2004) Santa Fe, NM, April 2004.

Nicholas Dingle obtained an M.Sc. degree with Distinction in Computing Science from Imperial College London in 2001 and is currently completing his Ph.D. thesis at the same institution. His research centres around techniques for the performance analysis of models of very large concurrent systems, with a particular focus on the use of parallel and distributed approaches for extracting response time densities and quantiles.

Peter Harrison is currently a Professor of Computing Science at Imperial College London, where he became a lecturer in 1983. He graduated at Christ's College Cambridge as a Wrangler in Mathematics in 1972 and went on to gain a Distinction in Part III of the Mathematical Tripos in 1979, winning the Mayhew prize for Applied Mathematics. He obtained his Ph.D. in Computing Science at Imperial College London in 1979. He has researched into analytical performance modelling techniques and algebraic program transformation for some twenty years, visiting IBM Research Centers for two summers in the last decade. He has written two books, had over 150 research papers published and held a series of both national and international research grants. The results of his research have been exploited extensively in industry, forming an integral part of commercial products such as Metron's Athene Client-Server capacity planning tool.

William Knottenbelt completed his B.Sc. (Hons) and M.Sc. degrees in Computer Science at the University of Cape Town in South Africa before moving to London in 1996. He obtained his Ph.D. in Computing from Imperial College London in February 2000, and was subsequently appointed as a Lecturer in the Department of Computing in October 2000. His research interests include stochastic performance modelling and parallel computing. He has also done consultancy work for a number of companies including ICL Fujitsu, LogicaCMG and iTouch PLC.