Distributed Solution of Large Markov Models Using Asynchronous Iterations and Graph Partitioning

Nicholas J. Dingle William J. Knottenbelt*

June 15, 2002

Abstract

We present a distributed approach for the steady state solution of large Markov models. We use asynchronous iterations to minimise processor idle time and graph partitioning techniques to minimise inter-processor communication. We demonstrate the scalability of our approach by solving a benchmark model for a number of large state space sizes on both a network of commodity PCs and a distributed memory parallel computer. The performance of our approach is contrasted with published results for an out-of-core solver.

1 Introduction

The steady state solution of Markov models is a challenge that often arises in performance engineering. Traditionally, performance statistics for high-level models (such as stochastic Petri nets and queueing networks) are derived by generating and then solving a Markov chain which corresponds to the model's behaviour at the state-transition level. From the Markov chain's steady state probability distribution, high-level performance measures such as throughput and mean buffer occupancy can be derived. However, workstation memory and compute power are often overwhelmed by the sheer number of states in the Markov chain.

For structurally unrestricted large scale models, the most effective computational approaches to date have been disk-based [DS98, KM02] and parallel [Kno00] methods. Both offer the ability to handle very large state spaces, but both have certain drawbacks which limit their performance. Disk-based methods are out-of-core solution techniques that read in matrix elements (and sometimes vector elements) from disk as needed. Although limited opportunities for caching sometimes exist through the reuse of matrix blocks, the speed of disk-based methods is essentially limited by the speed at which data can be read from disk so CPU utilisation is typically poor. On the other hand, parallel numerical methods can exploit the memory and processor capacity of several workstations but are constrained by the speed with which information (e.g. vector updates) can be exchanged between processors. High communication overhead is particularly a problem on commodity networks of workstations linked by Ethernet, where poor speedups and even slow downs are often observed. Parallel implementations may also suffer from load imbalance and interprocessor synchronisation overheads.

*Department of Computing Imperial College of Science, Technology and Medicine 180 Queen's Gate, London SW7 2BZ, United Kingdom. Email: {njd200,wjk}@doc.ic.ac.uk

We propose a distributed approach which seeks to improve performance by reducing the amount of communication necessary between the processors involved and removing the need for interprocessor synchronisation. Graph partitioning is used to divide the Markov model's generator matrix Q across the processors. This ensures that the number of off-diagonal matrix elements on each processor (i.e. those which require multiplication with non-local vector elements) is minimised, and therefore reduces the amount of communication needed. Further, asynchronous iterations are employed so that the processors are always fully utilised and are not required to wait for data to be exchanged.

The rest of this paper is organised as follows. Section 2 discusses the parallel solution of systems of linear equations using asynchronous iterations and shows how this can be applied to Markov models. Section 3 introduces graph partitioning and explains the advantages it offers in the proposed parallel computation scheme. Section 4 details the implementation of our algorithm, numerical results from which are given in Section 5. Finally, Section 6 concludes and discusses future work.

2 Parallel Solution of Systems of Linear Equations with Asynchronous Iterations

For a continuous-time Markov chain with an $n \times n$ transition rate matrix Q, the steady state probability distribution vector π can be computed by solving the equation:

$$\pi Q = 0$$

subject to

$$\sum_i \pi_i = 1.$$

The equation $\pi Q = 0$ can be rearranged as $Q^T \pi^T = 0$ which yields an expression of the form $Ax = b$ for which a number of well-known sequential iterative solution techniques exist (e.g. Jacobi, Gauss-Seidel, SOR, Conjugate Gradient Squared etc.) [Ste94]. When such calculations are conducted in parallel, iterative algorithms for solving the resulting systems of linear equations usually require an exchange of x-values at the end of each iteration. This enforced barrier synchronisation limits execution speed to that of the slowest processor.

Asynchronous iterative algorithms [FS99, KBS99] attempt to overcome this. Rather than exchange vector elements at the end of every iteration, processors send updates to each other every n iterations (in the implementation below we set n=5). These values are then used until the next update is received, meaning that some x-values will be out of date. This does delay the convergence of the solution, but as it is usually the case that the computation capacities of a network of processors exceeds that network's communication capacity it may be advantageous to increase the amount of computation performed while reducing communication.

We implemented a parallel block Successive Over-Relaxation (SOR) algorithm [Ste94] in our asynchronous steady state solver. In this scheme, each processor i is allocated n_i consecutive rows of the partitioned Q^T matrix (see Section 3 below for details of how these partitions are created) which are stored in local memory. The n_i elements of the vector x that correspond to the matrix rows assigned (denoted by x_i) are also stored in local memory. Fig. 1 illustrates how a sample partitioned matrix Q^T and vector x are divided between 4 processors.

At the beginning of an outer iteration, each processor calculates $y_i = \sum_{j \neq i} Q^T_{ij} x_j$ where the notation Q^T_{ij} indicates the jth matrix block of processor i ($0 \leq i, j < p$). The y_i vectors remain fixed for the rest of the outer iteration. Five inner iterations are then performed within the outer iteration, each consisting of a single SOR

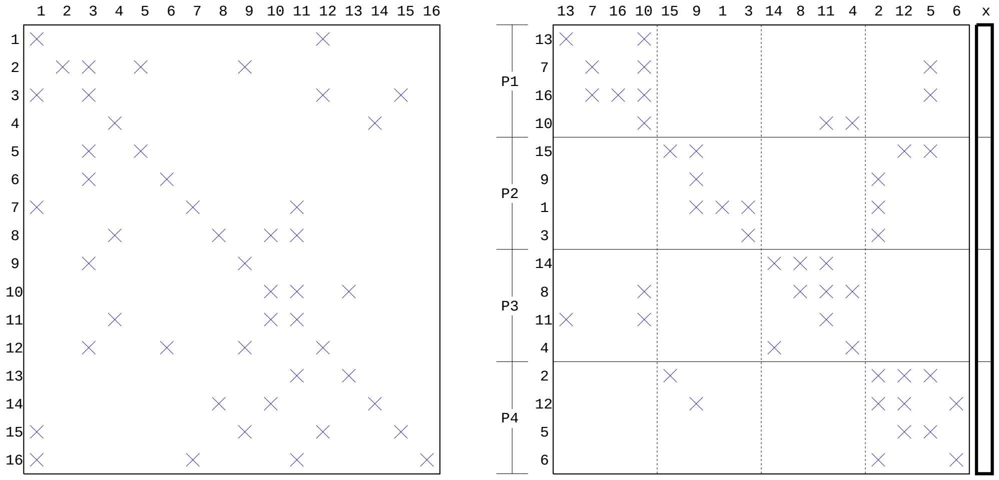

Figure 1: Original Q^T matrix (left) and partitioned Q^T matrix and x vector and corresponding row-striped mapping onto processors (right).

iteration using a locally determined relaxation parameter ω. After the fifth inner iteration, each processor transmits to every other processor any elements of x_i which that processor requires (from a list created when the processors were first initialised). These updates are performed asynchronously – processors are not required to wait for all others to reach the end of their inner iterations before sending updates but rather can simply transmit these updates and proceed with their computation. Also, every processor calculates the current convergence of its local x_i elements at this point and updates its ω value based on how well convergence is progressing. Execution then proceeds to the next outer iteration.

3 Graph Partitioning

Most iterative solution techniques for Markov chains are based on successive matrix-vector multiplications or very similar operations. In order to perform these multiplications efficiently in parallel it is necessary to map the non-zero elements of Q^T onto processors in such a way that the communication between processors is minimised and the computation load is balanced (although the latter is not such a crucial factor when asynchronous iterations are employed). In the following, we aim to produce an near-optimal row-striped allocation of the non-zero elements of Q^T to processors by applying graph-based partitioning.

Allocating the n rows of Q^T to p processors is known to be equivalent to a p-way partitioning of an undirected graph with n vertices [KK98b]. This graph is constructed as follows: graph nodes i and j are connected by an arc of weight one if either $Q^T_{ij} \neq 0$ or $Q^T_{ji} \neq 0$. Graph nodes i and j are connected by an arc of weight two if both $Q^T_{ij} \neq 0$ and $Q^T_{ji} \neq 0$ [CA99].

This graph is partitioned with the goal of minimising the number of edges which are *cut* (that is to say which span two partitions) whilst maintaining an acceptable load-balance of non-zero elements across the partitions. Edges which are cut correspond to elements which must be exchanged between the processors during a parallel matrix-vector multiplication, so that minimising the edge-cut is an attempt to minimise the amount of communication required between processors.

The problem of producing an optimal p-way partition is known to be NPcomplete [KK98b]. However, there exist a number of tools which implement heuristic algorithms to calculate good decompositions, for example Chaco [HL95] and MeTiS [KK98a]. A library of parallel MeTiS routines called ParMeTiS [KK96] is also available. Using these routines, we implemented a parallel graph partitioner which does not require the entire Q^T matrix to be stored on a single processor at any point in its execution – the memory consumed on each of the processors in the partitioning is inversely proportional to the number of processors employed.

Graph partitioning offers a particular advantage when employed in conjunction with asynchronous iterations. The effect of partitioning a matrix using graph partitioning is to cluster the majority of the non-zero entries (typically in excess of 90%) along the diagonal (as can be seen in the partitioned matrix of Fig. 1). These are the elements which will be multiplied with the portion of the x-vector which is stored locally, whilst the off-diagonal elements are multiplied with vector elements which are updated by other processors. This is beneficial for the convergence behaviour of the implementation.

4 Parallel Algorithm and Implementation

The calculation of the steady state probabilities of a large Markov model begins with a high-level model specified in the DNAmaca Markov Chain Analyser interface language [Kno96, Kno00]. This permits the specification of queueing networks, stochastic Petri nets, stochastic Process Algebra and other formalisms which can be mapped onto Markov chains. A probabilistic hash-based state generator [KMHK98] uses the high-level description to generate the rate matrix Q of the model's underlying Markov chain. This matrix is then transposed to produce Q^T and the corresponding graph is constructed as described in Section 3. This graph is partitioned by a parallel graph partitioner (implemented using the ParMeTiS library [KK96]) and the resulting mapping is applied to Q^T to divide its rows between the processors.

The distributed steady state calculator is implemented in C++ and uses the Parallel Virtual Machine (PVM) [GBD$^+$94] library for inter-processor communication. This was chosen in preference to the standard Message Passing Interface (MPI) [GLS94] because PVM permits truly asynchronous, buffered communication – i.e. there is no requirement for a matching receive to have been posted when a message is sent.

We hoped to adopt a multi-threaded architecture for our steady state solver which would operate as follows. Each processor reads in the rows of the matrix Q^T which correspond to its allocated partition into memory and then spawns a new thread (implemented using the POSIX standard `pthread` library). This thread is responsible for computation while the original process is responsible for handling communication between the processors. The communication process must handle a variety of messages: it must request and send vector updates as well as collect and transmit convergence data. When all calculations have converged the calculation threads must be told to stop and the final probability vectors collected on the master machine.

However, we discovered that the overhead incurred in context switching between the two threads degraded the performance of the solver dramatically. Instead we implemented a non-threaded version where calculation is suspended after a fixed number of multiplications in order to check if any messages have been received and to handle those which have. This approach exploits the asynchronous message handling provided by PVM.

	AP3000			PC Cluster		
p	time (s)	speedup	efficiency	time (s)	speedup	efficiency
1	1182.5	1.00	1.000	363.2	1.00	1.000
2	853.3	1.39	0.695	328.2	1.11	0.555
4	409.2	2.89	0.723	193.3	1.88	0.470
8	213.4	5.54	0.693	150.3	2.42	0.303
16	135.9	8.70	0.544	134.3	2.70	0.169
32	83.8	14.11	0.441	112.4	3.23	0.101

Table 1: Run time, speedup and efficiency for p-processor steady state solution for the FMS model with k=7. Results are presented for an AP3000 distributed memory parallel computer and a PC cluster.

5 Results

We demonstrate the accuracy and scalability of our implementation by performing an analysis of the well-known Flexible Manufacturing System (FMS) Generalised Stochastic Petri Net model [CT93]. Setting k (the number of unprocessed parts in the system) to 7 results in the underlying Markov chain of the GSPN having 1 639 440 tangible states and produces 13 552 968 off-diagonal entries in its generator matrix Q.

Table 1 summarises the performance of the implementation on a distributed memory parallel computer and a cluster of workstations. The parallel computer is a Fujitsu AP3000 which has 60 processing nodes (each with an UltraSparc 300MHz processor and 256MB RAM) connected by a 2D wraparound mesh network. This network uses wormhole routing and has a peak throughput of 520Mbps. The PC cluster is comprised of 32 Athlon 1.4GHz PCs each with 512MB RAM and linked together by a 100Mbps switched Ethernet network. The distributed run time was measured from the beginning of the first outer iteration until the final result was assembled on the master processor. These results can be compared with an out-of-core solution technique which took 629.8 seconds to calculate the steady state solution for this model on a single 1.4GHz workstation.

Corresponding graphs of the run time and of the speedup and efficiency are presented in Figures 2 and 3 respectively. We define the speedup obtained by executing the solver on p processors to be the run time for a sequential execution (p=1) divided by the run time for the p-processor execution. The efficiency for p processors is the speedup divided by the number of processors.

The speedups and efficiencies obtained from executing our implementation on the AP3000 are good. For the 16-processor execution, the observed efficiency of 54% is higher than the 41% previously achieved for a naively-partitioned synchronous steady state solution of the same model on the same hardware [KH99]. We also note that a reasonable speedup is achieved on the PC cluster, although the trend is much shallower than that for the AP3000.

In order to compare the performance of algorithm against published out-of-core results we also tested it on the FMS model with k=8. The underlying Markov chain in this case contains 4 459 455 states. Using a cluster of 32 PC workstations of the specification described above the result was calculated in 417.4 seconds. This compares very favourably with the 18 753 seconds required to solve the same model by an out-of-core implementation [KM02]. The terminating condition for the convergence of our algorithm was much stricter than that of the out-of-core solver – our calculations continued until the convergence was less than 10^{-10}, whilst for the out-of-core solver the threshold was 10^{-6}. It must be noted, however, that the processor used in the out-of-core solver (a single UltraSparc-II 360Mhz machines

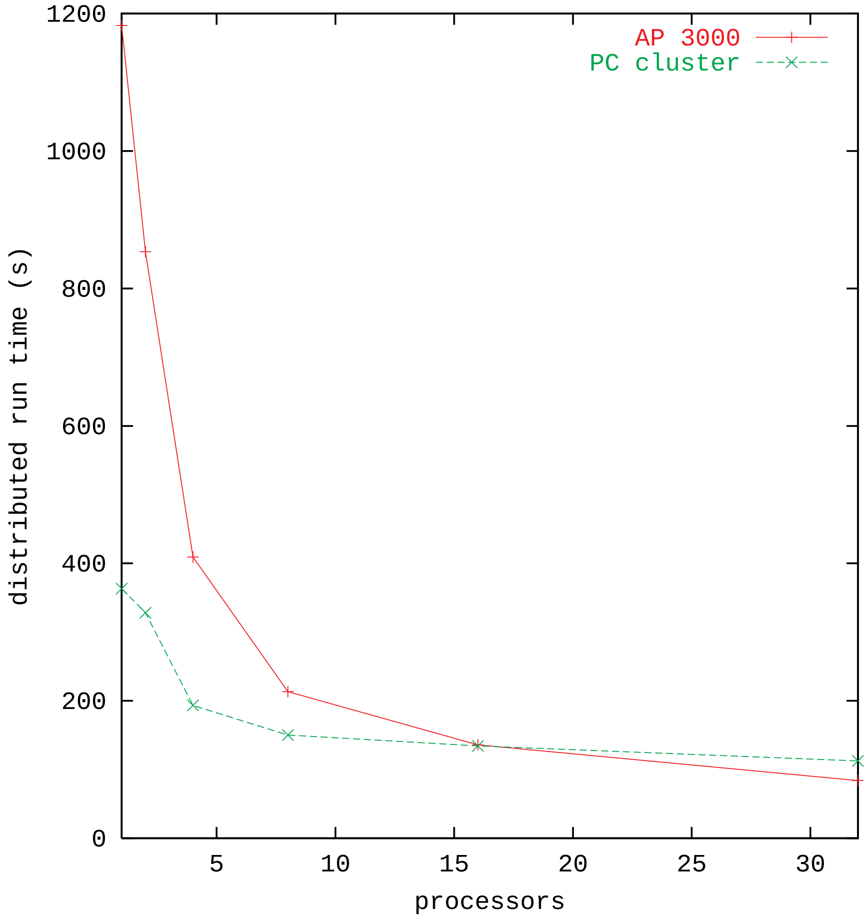

Figure 2: Distributed run time for the FMS model with k=7 on the AP3000 and a PC cluster.

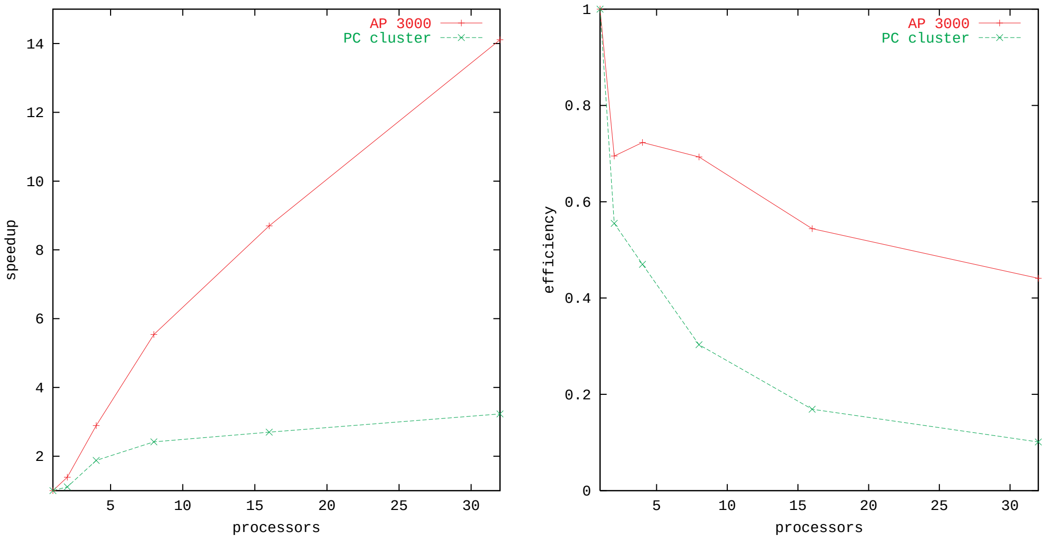

Figure 3: Speedup (left) and efficiency (right) for the FMS model with k=7 on the AP3000 and a PC cluster.

with 128MB RAM) was slower than the workstations used in our evaluation, but this would probably not have been a major factor in the relative performance difference as the out-of-core solver would have been limited by disk throughput rather than processor speed.

6 Conclusion

The results presented in this paper are taken from an early implementation and we are continuing to refine our approach. Nevertheless, the initial results presented here suggest that our approach has much to commend it. It has been shown to have good scalability and also performs better than comparable out-of-core solution techniques. A key development will be to increase the capacity of our technique to handle even larger state spaces – indeed, the choice of k=8 for our large example was due to limitations in the techniques employed to prepare Q^T for use in our steady state solver, rather than any limitation of the solver itself. Work on rectifying this has already begun.

7 Acknowledgements

The authors would like to thank the Imperial College Parallel Computing Centre for the use of the AP-3000 distributed memory parallel computer.

References

[CA99] U.V. Catalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.

[CT93] G. Ciardo and K.S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[DS98] D.D. Deavours and W.H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33(1):67–84, June 1998.

[FS99] A. Frommer and D.B. Szyld. On asynchronous iterations. Research Report 99-5-31, Department of Mathematics, Temple University, Philadelphia, USA, May 1999. To appear in *Journal of Computational and Applied Mathematics.*

[GBD$^+$94] 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.* MIT Press, Cambridge, Massachussetts, 1994.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, Massachussetts, 1994.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the ACM/IEEE Supercomputing Conference.* ACM/IEEE, December 1995.

[KBS99] D. Szyld K. Blathras and Y. Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *Journal of Parallel and Distributed Computing*, 58(3):446–465, September 1999.

[KH99] W.J. Knottenbelt and P.G. Harrison. Distributed disk-based solution techniques for large Markov models. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99)*, pages 58–75, Zaragoza, Spain, September 1999.

[KK96] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report #96–036, University of Minnesota, 1996.

[KK98a] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report #95-035, University of Minnesota, 1998.

[KK98b] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998. URL http://www.cs.unm.edu/∼karypis.

[KM02] M. Kwiatowska and R. Mehmood. Out-of-core solutions of large linear systems of equations arising from stochastic modelling. In *Proceedings of Process Algebra and Performance Modelling*, Copenhagen, July 25th–26th 2002.

[KMHK98] W.J. Knottenbelt, M.A. Mestern, P.G. Harrison, and P.S. Kritzinger. Probability, parallelism and the state space exploration problem. In *Lecture Notes in Computer Science 1469: Proceedings of the 10th International Conference on Modelling, Techniques and Tools (TOOLS '98)*, pages 165–179, Palma de Mallorca, Spain, September 1998. Springer Verlag.

[Kno96] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.

[Kno00] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, February 2000.

[Ste94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.