Distributed Disk-based Solution Techniques for Large Markov Models

William Knottenbelt* Peter Harrison

Department of Computing, Imperial College of Science Technology and Medicine
180 Queens Gate, London SW7 2BZ, United Kingdom

March 2, 1999

Abstract

Very large Markov chains often arise from stochastic models of complex real-life systems. In this paper we investigate disk-based techniques for solving such Markov chains on a distributed-memory parallel computer. We focus on two scalable numerical methods, namely the Jacobi and Conjugate Gradient Squared (CGS) algorithms.

The critical bottleneck in these methods is the parallel sparse matrix-vector multiply operation. By exploiting the data locality typically found in automatically generated state-transition-rate matrices, we develop an efficient matrix-vector multiply kernel that is characterised by low per-processor memory usage, low communication cost, and good load balance. At a slightly higher memory cost, the kernel also allows for the overlapping of communication and computation.

We describe a distributed software architecture which makes use of two processes per node to allow for the overlapping of disk I/O with computation and communication. By embedding our matrix-vector multiply kernel in this architecture, we implement a high-performance disk-based solver on a 16-node Fujitsu AP3000 computer. We demonstrate results showing good speedups and the capability to analyse very large models with over 50 million states and over 500 million transitions.

1 Introduction

Several formalisms can be used to specify performance models, for example stochastic Petri nets, stochastic process algebras and queueing networks. Traditionally, performance measures for such models are derived by generating and solving a Markov chain corresponding to the model's state-level transition behaviour. However, workstation memory and compute power are frequently overwhelmed by the sheer size of the Markov chain's state space and infinitesimal generator matrix.

There are several techniques which attempt to cope with this state space explosion or "largeness" problem. Most of these techniques avoid explicit storage of the infinitesimal

*William Knottenbelt is the Beit Fellow for Scientific Research in the Department of Computing at Imperial College. Peter Harrison is a professor in the same department. Email: {wjk, pgh}@doc.ic.ac.uk.

generator matrix by exploiting structure in the model [Buc95, Kem95]. Here, however, we will consider the general case where no symmetry, hierarchy or other structure is assumed.

By far the most promising generally applicable technique for the large-scale solution of Markov models on a single processor is the "disk-based" solution proposed by Deavours and Sanders [DS98]. Their Block Gauss-Seidel (BGS) solver stores the infinitesimal generator matrix on disk and maintains high disk throughput using a system of two cooperating processes which perform disk I/O and computation concurrently. The memory usage of the BGS solver is low with the main requirement being the space for the solution vector. In this way, systems of 10 million states and 100 million transitions can be solved on a workstation with only 128MB RAM.

With the availability of distributed-memory computers and high-speed workstation clusters, much attention has been focussed on methods for distributed and parallel state space generation [CCM95, AH97, CGN98]. Recent developments mean that it is possible to generate very large state spaces of over 50 million states and 500 million transitions on a 12-node distributed-memory parallel computer in under half an hour [KMHK98]. A single workstation is inadequate to solve models of this scale – the amount of computation required is vast and the space required for the solution vector alone requires more memory than is available on most workstations.

Corresponding distributed techniques which leverage the compute power, memory and disk space of several processors are therefore needed to solve these models. This is the focus of the present study, which considers disk-based solution techniques for distributed-memory computers. In particular we consider two numerical methods that are suited to parallel implementation: the Jacobi and Conjugate Gradient Squared (CGS) algorithms. We discuss opportunities for parallelism and show how the memory requirements of the CGS algorithm can be reduced at the cost of extra disk space.

Achieving good parallel performance from sparse matrix problems is a challenge which often arises in scientific computing. The situation is particularly difficult when the bandwidth is low. This is often the case with Markov models where there are usually only a limited number of events that can occur in each state, leading to a limited number of successor states. This problem manifests itself in the sparse matrix-vector multiply operation which lies at the core of the Jacobi and CGS algorithms. However, by exploiting the structure induced by breadth-first search state generation algorithms we develop an efficient matrix-vector multiply kernel which exhibits low memory use, low communication load and good load balance.

We combine our kernel with a high-performance distributed software architecture which makes use of two processes per node to maximise the overlapping of disk I/O with communication and computation. We have implemented the resulting solver on a distributed-memory parallel computer with 16 nodes, each of which has a 300MHz processor and 256MB RAM. We demonstrate its effectiveness of our tool by solving Markov chains of up to 50 million states and 500 million transitions.

The rest of this paper is organised as follows. Section 2 outlines the Jacobi and Conjugate Gradient Squared algorithms and considers opportunities for parallelism. Section 3 shows how sequential and distributed breadth first generators induce a structure on the infinitesimal generator matrix which can be used to develop an efficient matrix-vector multiply kernel. Section 4 presents a software framework for a high-performance distributed disk-based Markov solver. Results from an implementation which embeds the matrix-vector

multiply kernel in this framework are presented in Section 5. Section 6 concludes and considers future work.

2 Scalable Numerical Methods

Solving a continuous time Markov chain with n states corresponds to solving the set of steady-state equations of form:

$$\pi Q = 0, \quad \sum \pi_i = 1$$

where Q is the $n \times n$ infinitesimal generator matrix and π is the n-element steady-state solution vector. An equivalent formulation is $Q^T \pi^T = 0$ which allows the use of general algorithms for solving $Ax = b$. Note that in general, we can assume $A_{ii} = Q^T_{ii} = 1$ without loss of generality, since $Q^T \pi^T$ can be transformed into By where $B = Q^T D^{-1}$ and $y = D\pi^T$ with $D = \text{diag}(Q^T_{11}, Q^T_{22}, \ldots, Q^T_{nn})$. π^T is then easily obtained as $D^{-1}y$.

A broad spectrum of sequential solution techniques are available for solving steady-state equations [Ste94]. 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 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, this is a combinatorial problem of exponential complexity and obtaining suitable orderings for very large matrices is infeasible.

Most classical iterative methods, such as Gauss-Seidel and 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 and will be used as a base case for comparison.

Krylov subspace methods [Wei95] 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. The most recently developed algorithms (CGS [Son89], BiCGSTAB [Vor92] and TFQMR [Fre93]) are also suitable for a disk-based implementation since they access A in a predictable fashion and do not require multiplication with A^T. Compared to classical iterative methods, however, Krylov subspace techniques have high memory requirements. We select CGS for our study because it requires the least memory of these methods; further we devise a scheme for reducing the total memory requirement (across all processors) from 7 n-vectors to just 3 by storing intermediate vectors on disk.

JACOBI ALGORITHM

1. Initialise
   - Q^T is given and $x^{(0)}$ is an initial guess at the solution vector.
   - $r^{(0)} = -Q^T x^{(0)}$
   - $k = 0$
2. Iterate
   - **while** $||r^{(k)}||_\infty / ||x^{(k)}||_\infty > 10^{-10}$ **do**

     $k = k + 1$

     for $i = 0$ **to** $n - 1$ **do**

     $x_i^{(k)} = r_i^{(k-1)} / Q_{ii}^T + x_i^{(k-1)}$

     $r^{(k)} = -Q^T x^{(k)}$
3. Normalise x.

Figure 1: The Jacobi method [KGGK94].

2.1 Jacobi Method

Jacobi's method is a simple iterative method which is based on the observation that a solution to $Ax = b$ satisfies:

$$x_i = (b_i - \sum_{i \neq j} A_{ij} x_j) / A_{ii}$$

This suggests the iterative form

$$x_i^{(k+1)} = (b_i - \sum_{i \neq j} A_{ij} x_j^{(k)}) / A_{ii}$$

where k starts at 0 and $x^{(0)}$ is an initial guess at the solution vector. We can rework this equation in terms of the residual $r = b - Ax$ by observing that $r_i^{(k)} = b_i - \sum_{j=0}^{n-1} A_{ij} x_j^{(k)}$, so

$$x_i^{(k+1)} = (b_i - \sum_{i=0}^{n-1} A_{ij} x_j^{(k)} + A_{ii} x_i^{(k)}) / A_{ii} = r_i^{(k)} / A_{ii} + x_i^{(k)}$$

The algorithm based on this formulation appears in Fig. 1. Since calculations of the $x_i^{(k)}$'s are independent, vectors can be distributed and equation updates performed in parallel. There is one matrix-vector product which may also be performed in parallel, although this requires communication. Total storage requirements across all processors amounts to 3 n-vectors.

The stopping condition we choose is $||r^{(k)}||_\infty / ||x^{(k)}||_\infty < \epsilon$ with $\epsilon = 10^{-10}$. This is a good measure of the quality of the solution relative to the size of elements in the unnormalised solution vector.

2.2 Conjugate Gradient Squared Algorithm

CGS Algorithm

1. Initialise
   - Q^T is given
   - $x^{(0)}$ is an initial guess at the solution
   - $r^{(0)} = -Q^T x^{(0)}$
   - $\hat{r}^{(0)} = r^{(0)}$
   - $\rho^{(0)} = 1$
   - $p^{(0)} = q^{(0)} = \underline{0}$
   - $k = 0$

1	2	3
	x	
r	x_W	m
r	$\hat{r}_W$	-
r	p_W	q_W

2. Iterate
   - **while** $||r^{(k)}||_\infty / ||x^{(k)}||_\infty > 10^{-10}$ **do**

     $k = k + 1$

     $\rho^{(k)} = \hat{r}^{(0)} \cdot r^{(k-1)}$

     $\beta = \rho^{(k)} / \rho^{(k-1)}$

     $u = r^{(k-1)} + \beta q^{(k-1)}$

     $p^{(k)} = u + \beta(q^{(k-1)} + \beta p^{(k-1)})$

     $v = Q^T p^{(k)}$

     $\alpha = \rho^{(k)} / (\hat{r}^{(0)} \cdot v)$

     $q^{(k)} = u - \alpha v$

     $u = u + q^{(k)}$

     $x^{(k)} = x^{(k-1)} + \alpha u$

     $r^{(k)} = -Q^T x^{(k)}$

1	2	3
r	$\hat{r}^R$	
r	q^R	u
p^R	q	u_W
p_W	v	m
$\hat{r}^R$	v	
q	v	u^R
q_W		u
	x^R	u
r	x_W	m

3. Normalise x.

Figure 2: The CGS algorithm [Son89].

The Conjugate Gradient Squared (CGS) algorithm [Son89] is a generalisation of the classical Conjugate Gradient method in that it allows for a general non-symmetric matrix A instead of requiring A to be symmetric positive definite. The algorithm is shown in Fig. 2. Greek letters (α, β and ρ) represent scalar values while Roman letters (p, q, r etc.) represent n-vectors (with the exception of the iteration counter k). CGS performs two matrix-vector multiplications, 6 vector updates and 2 dot products per iteration. These operations give much scope for parallelism since vectors can be completely distributed and calculation can proceed largely independently. Communication is only required for the matrix-vector multiplication, the dot products and the calculation of vector norms involved in the convergence test.

The memory requirements of the CGS algorithm are high, since storage is required for a total of 7 n-vectors across all processors (p, q, r, u, v, x and m which is a vector required by the distributed vector-multiply). Not all of these vectors are used at the same time, however, and it is possible to reduce the total memory requirement to just 3 n-vectors at the cost of writing some intermediate vectors to disk. The schedule which achieves this

goal is shown to the right of the CGS algorithm in Fig. 2. The notation x^R indicates that vector x is read from disk at the start of an operation and x_W indicates the vector is written to disk at the end of the operation.

The stopping condition is the same as for the Jacobi algorithm. Since the convergence of the CGS algorithm is often erratic, this avoids false convergence problems associated with stopping conditions based on $||x^{(k+1)} - x^{(k)}||_\infty$, and allows for fair comparison of the two algorithms.

3 Distributed sparse matrix-vector multiply kernel

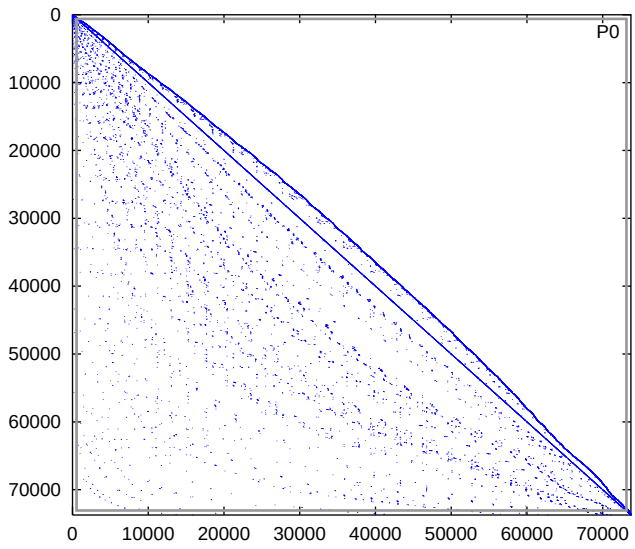
The sparse matrix-vector multiply operation $Q^T x$ forms the core of both the Jacobi and CGS algorithms. Consequently an efficient implementation of this kernel is central to obtaining good performance. Ideal attributes of the kernel include low communication load, good load balance, good scalability and the ability to overlap communication and computation. In addition, per-processor memory requirements should be kept as low as possible since storing vectors of double precision floating point numbers is expensive (usually eight bytes each).

In the following sections, we study the structure of the infinitesimal generator matrix Q as typically produced by a large class of sequential and distributed state space generation tools. We then consider various state reordering strategies which can be used during matrix tranposition to obtain an effective data distribution for Q^T. Finally, we present an outline of an efficient matrix-vector multiply kernel.

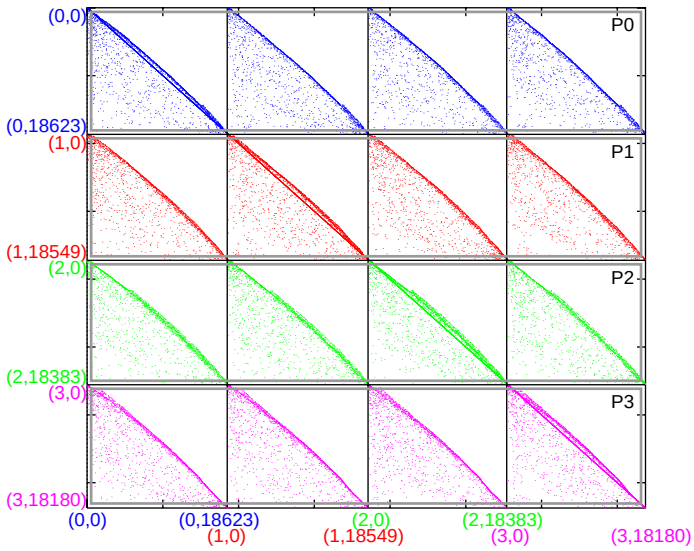
3.1 Infinitesimal Generator Matrix structure

Automated state space generation tools are widely used to map structurally unrestricted high-level models onto their underlying state spaces and infinitesimal generator matrices. Typically this mapping is performed using a sequential breadth-first search (BFS) traversal which assigns unique state sequence numbers to states in the order in which they are encountered. Fig. 3(a) demonstrates how a sequential BFS generator induces a (mostly) lower-diagonal structure on the resulting infinitesimal generator matrix Q.

Distributed state space generators [CCM95, CGN98, KMHK98] use hash functions to partition states across nodes so that each node is responsible for exploring a portion of the state space and for constructing a portion of the generator matrix Q. Each node performs a BFS-like exploration of a local state queue in a manner similar to the sequential algorithm. Newly discovered states are passed to their "owner" processors where they are inserted into the local queue and assigned a local state sequence number. Given p processors each of which generates n_i states, states in this scheme are identified by a pair of integers (i, j) where i $(0 \leq i < p)$ is the node number of the host processor and j $(0 \leq j < n_i)$ is the local state sequence number. Fig 3(b) illustrates the resulting structure induced by a distributed BFS generator.

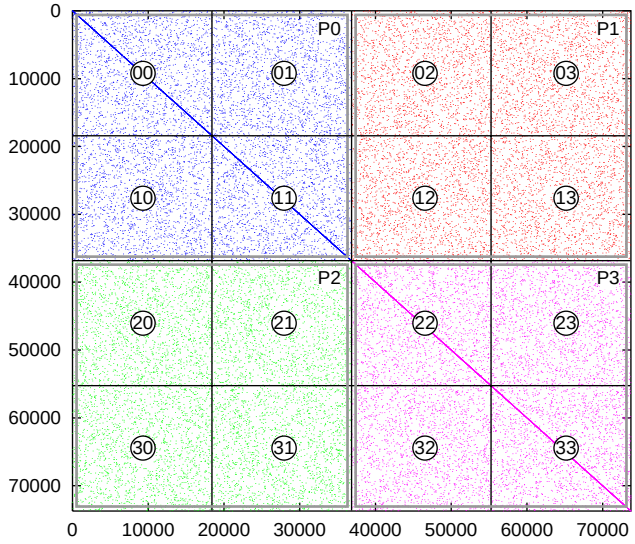


(a) Sequential BFS generator.

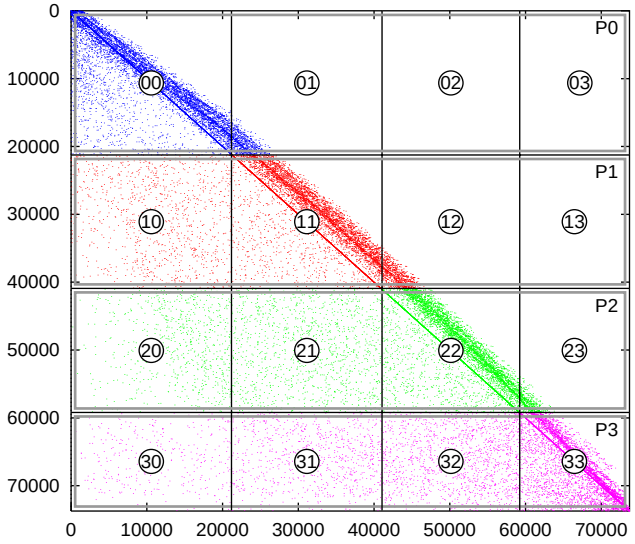


(b) Distributed BFS generator (4 processors).

Figure 3: The non-zero structure of the infinitesimal generator matrix Q as induced by sequential and distributed breadth-first state generators. The matrices are derived from a queueing Petri net model of a telecommunications protocol with an underlying Markov chain of 73 735 states and 295 591 transitions.



(a) Random remapping with block-checkerboard assignment of processors.



(b) Approximate BFS remapping with row-wise assignment of processors such that the number of non-zeros allocated to each processor is the same.

Figure 4: The structure of Q^T after state reordering and corresponding processor assignments.

3.2 Matrix reordering strategies

3.2.1 Random reordering

The most efficient matrix-vector multiply algorithms for *dense* matrices are those based on a block-checkerboard partitioning in which processors are assigned $n/\sqrt{p} \times n/\sqrt{p}$ blocks of matrix elements [KGGK94]. Such algorithms rely on a balanced distribution of elements across processors and regular interprocessor communication that can be conducted in parallel. In general, however, sparse matrices have an unbalanced non-zero structure so computation time is determined by the block with the largest number of non-zeros. Ogielski and Aiello overcome this problem by showing that randomly permuting the rows and columns of a sparse matrix produces a well-balanced block allocation with high-probability [OA93]. The resulting matrix can then be used to good effect with well-known general algorithms for sparse parallel matrix-vector multiplication [LG93, LPG94, HLP95].

In our case, the same random scattering effect can be achieved by using a pseudo-random function $f(i,j) \rightarrow (0,\ldots,n-1)$ to assign state (i,j) to a unique global state number according to the mapping:

$$f(i,j) = (c_1 * (\sum_{k=0}^{i-1} n_k + j) + c_2) \bmod n$$

where n_i is the number of states generated by node i, c_1 is a large prime and c_2 is an arbitrary offset. The global state number can then be used to partition the states over the nodes in a straightforward fashion. Fig. 4(a) shows the resulting layout of Q^T after the application of this mapping and a corresponding block-checkerboard assignment of processors.

3.2.2 Approximate BFS reordering

The random remapping described above allows for the application of well-known efficient algorithms but suffers from high communication cost. One approach to alleviating this bottleneck is to reorder the states of Q^T across processors to maximize data locality and minimise communication. This goal maps directly onto a p-way weighted graph partitioning problem. This problem involves subdividing the vertices of a weighted graph into p equal partitions such that the number of edges that straddle partitions is minimised and the sum of the vertex weights in each partition is the same [KK98]. In our case the states in Q^T correspond to the vertices of the graph, the transitions between states constitute the edges and the number of non-zeros in a row are the vertex weights. This problem is NP-complete. We can, however, obtain considerable data locality by using a rapid mapping which exploits the structure evident in Fig. 3(b). In particular, if we assign state (i,j) to a global state number given by the function

$$f(i,j) = n - \sum_{k=0}^{i} \min(j+1, n_k) - \sum_{k=i+1}^{p-1} \min(j, n_k)$$

we obtain the BFS-like structure for Q^T shown in Fig. 4(b). It is then a straightforward task to assign blocks of consecutive states to processors such that the number of non-zeros

allocated to each processor is equal. This results in a row-wise allocation of matrix-blocks to processors, as shown in Fig. 4(b).

3.3 Kernel algorithm

We now outline a disk-based distributed sparse vector-matrix multiply kernel. We assume that the states of Q^T have been reordered according to the approximate BFS mapping described above, resulting in an allocation of s_i states to processor i. We will use the notation Q_{IJ} to indicate the jth matrix block of node i ($0 \leq i, j < p$). x_I will be used to denote the distributed portion of vector x of length s_i allocated to node i.

```
function multiply-kernel (Q^T_{I*}: matrix[s_i][n], x : vector[s_i]) : vector[s_i]

var     y       :   vector[s_i]
        m       :   vector[max_i(s_i)]
        j, k, p :   integer

begin
    y = 0
    for k = 0 to p − 1 do begin
        j = (i + k) mod p
        if Q^T_{IJ} is empty continue
        if i ≠ j do begin
            m = request-subvector(j)
            y = y + disk-multiply(Q^T_{IJ}, m)
        end else
            y = y + disk-multiply(Q^T_{II}, x)
    end
    serve-requests(x)
    return y
end
```

Figure 5: Distributed sparse matrix-vector multiply kernel for node i.

The algorithm for node i which performs $y_I = Q^T_{I*}x_I$ is outlined in Fig. 5. Node i begins by multiplying matrix block Q^T_{II} with x_I. This is performed by the procedure disk-multiply which reads blocks of non-zero elements from disk as necessary. Then, for each $j \neq i$ and non-empty block Q^T_{IJ}, the node calls request-subvector(j) which requests and receives from node j the subvector m to be multiplied with that block. To minimize communication, the subvector m contains only the sequence of elements in x_J which are actually referenced by the computation of $Q^T_{IJ}x_J$.

Nodes may use a dedicated thread to detect and service incoming requests for elements of x_J, thus allowing communication to proceed in tandem with the computation. Alternatively, in the case of a thread-unsafe message passing library, non-blocking probe and send

operations can be used to achieve the same effect. Any requests that remain outstanding are processed by the serve-request procedure.

The kernel algorithm as presented above blocks while waiting for remote subvectors. This can be avoided by taking further advantage of non-blocking communication primitives. In particular, during the initial multiplication $Q_{II}^T x_I$, node i can request the subvector required by the subsequent block $(i+1) \bmod p$. This subvector can be received into m using a non-blocking receive operation. At the cost of an extra vector of length $\max_i(s_i)$, this procedure can be extended to the remaining subblocks, thus reducing waiting time further by allowing for the complete overlap of communication and computation.

4 Tool architecture

This section describes a high-performance architecture for a distributed disk-based Markov Chain solver that makes use of our matrix-vector multiply kernel.

The limiting factor governing the computation speed of disk-based methods is usually disk throughput. This is especially the case with very large matrix files where operating system file caching is likely to be ineffective, resulting in poor disk throughput. It is therefore important for nodes to be able to overlap disk I/O and computation to achieve maximum efficiency. To solve this problem, Deavours and Sanders propose a two-process architecture which they use in their sequential disk-based Block Gauss-Seidel (BGS) solver [DS98]. We adapt this architecture to the distributed case, where the approach has the added benefit of allowing communication to proceed in parallel with disk I/O.

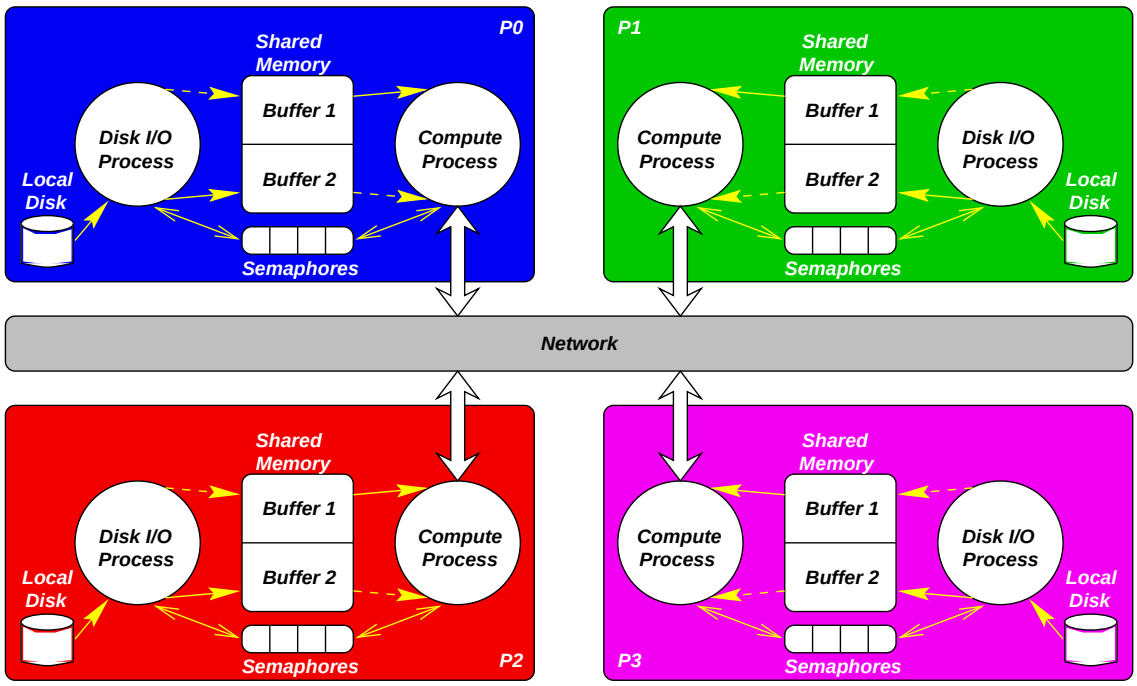


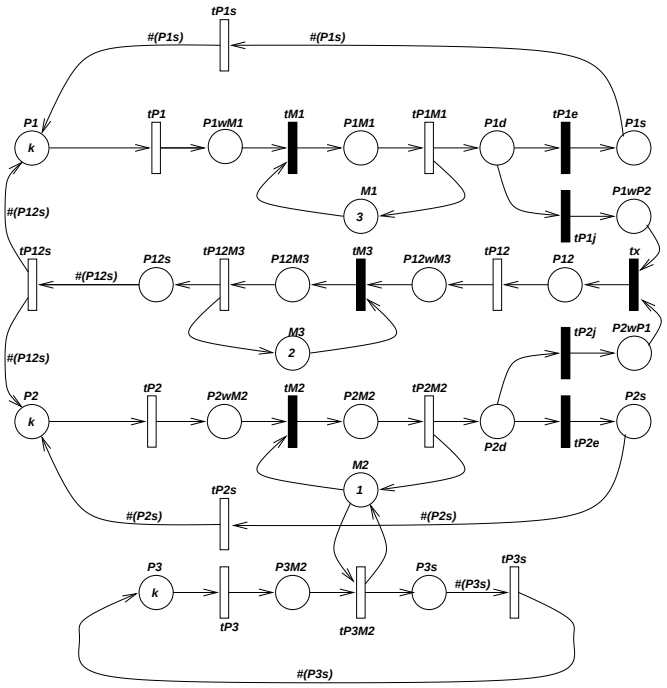
Figure 6: Tool architecture

Fig. 6 shows the architecture. 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 the 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. Interested readers are invited to consult [DS98] for a thorough exposition of this architecture and its benefits in a single processor context.

5 Results

We have implemented a distributed disk-based Markov solver which uses the kernel described in Section 3.3 and the software architecture outlined in Section 4. The solver is written in C++ and uses the Message Passing Interface (MPI) [GLS94] standard so it is portable to a wide variety of parallel computers and workstation clusters. The results presented here were obtained on a Fujitsu AP3000 distributed-memory parallel computer with 16 processing nodes [ITS97]. Each node runs the Solaris operating system and has a 300MHz UltraSparc processor, 256MB RAM and a 4GB local disk with uncached throughput of 6MB/s. The nodes are connected by a high-speed network with cut-through routing and a peak throughput of 50MB/s.



(a) Petri net representation

k	states (n)	transitions
1	54	155
2	810	3 699
3	6 520	37 394
4	35 910	237 120
5	152 712	1 111 482
6	537 768	4 205 670
7	1 639 440	13 552 968
8	4 459 455	38 533 968
9	11 058 190	99 075 405
10	25 397 658	234 523 289
11	54 682 992	518 030 370

(b) States and transitions in the underlying Markov chain

Figure 7: The FMS Generalised Stochastic Petri net [CT93]

To test our solver over a range of problem sizes, we consider the 22-place GSPN model of a flexible manufacturing system shown in Fig 7(a). This model, which we refer to as the

FMS model, was originally presented in detail in [CT93]. A detailed understanding of the model is not required, except to note that the model has a parameter k (corresponding to the number of initial tokens on places $P1$, $P2$ and $P3$) and that as k increases, so does the number of states n and the number of transitions in the state graph (see Fig. 7(b)). We use the same transition rates (many of them state-dependent) as given in [CT93].

The state spaces and infinitesimal generator matrices for the models were generated using the distributed state generation algorithm presented in [KMHK98]. The generator matrices were then remapped in a distributed fashion according to the reordering presented in Section 3.2.2. Both of these steps are rapid relative to the time taken for solution; 16 processors generate and remap the $k = 11$ (54 million states) case in under an hour of real time.

The resulting matrix blocks require 6 bytes of disk space per non-zero element – 4 for the column index and 2 used as in index into a vector of transition rates. The largest model requires about 18000 distinct transition rates so this approach is more economical than using 8 bytes to store each rate as a double precision number. The number of non-zero entries in each block row is also stored; one byte per block row is adequate since the bandwidth of the matrix is low (about 10). The latter information may be stored in memory for rapid access, or, for extremely large models, read in from disk during matrix-vector multiplication.

Table 1 presents the execution time (defined as maximum processor run-time) in seconds required for the distributed solution of models using the CGS and Jacobi methods. The models range in size from $k = 4$ (35 910 states) to $k = 11$ (54 million states) and runs are conducted on 1, 2, 4, 8, 12 and 16 processors.

The number of iterations and the per-node memory requirement for each run is also shown. The number of CGS iterations varies slightly with p whereas the number of Jacobi iterations remains constant. This occurs because the uniformly distributed starting vector $x^{(0)}$ used to initialise the Jacobi method is unsuitable for CGS since CGS's starting vector should ideally not be close to the final solution. Consequently we use a randomly-generated starting vector $x^{(0)}$ for CGS, with each processor using a different random seed.

Fig. 8 compares the convergence of the Jacobi method with that of the CGS algorithm for the $k = 7$ case in terms of the number of matrix-multiplications performed. The Jacobi method exhibits smooth but slow convergence, while the CGS algorithm exhibits rapid but erratic convergence. This trend also holds for the other results, with the CGS algorithm typically converging about 4 times faster than the Jacobi method.

The largest model that can be solved on a single processor is the case $k = 8$ (4.5 million states). Fig. 9 compares the average time taken per distributed CGS iteration for model sizes up to $k = 8$ and various numbers of processors. The graph shows a dramatic reduction in run-time as processors are added – for the case $k = 8$ 16 processors perform iterations at 25 times the speed of 1 processor. This superlinear speedup can be attributed to better caching of disk I/O, resulting in higher data throughput.

Fig. 10 shows the speedup and efficiency achieved by the CGS method for small models with $k < 7$ where variation due to caching effects does not play an important role. The speedup S_p for p processors is given by the run time of the sequential solution ($p = 1$) divided by the run time of the distributed solution with p processors. Efficiency for p processors is given by S_p/p. We see that that larger problem sizes produce better speedups,

		k=4	k=5	k=6	k=7	k=8	k=9	k=10	k = 11
$p = 1$	Jacobi time (s)	40.383	252.89	1160.0	4491.6				
	Jacobi iterations	1220	1500	1790	2095				
	CGS time (s)	16.817	111.25	479.09	2191.1	30974			
	CGS iterations	125	172	191	231	262			
	Memory/node (MB)	10.9	13.8	23.4	51.0	121.5			
$p = 2$	Jacobi time (s)	27.912	161.95	790.33	3535.9				
	Jacobi iterations	1220	1500	1790	2095				
	CGS time (s)	10.008	55.795	259.81	1082.1	13822			
	CGS iterations	125	154	189	217	279			
	Memory/node (MB)	10.5	11.9	17.00	31.3	68.0			
$p = 4$	Jacobi time (s)	24.574	113.51	633.99	2710.1				
	Jacobi iterations	1220	1500	1790	2095				
	CGS time (s)	7.0795	35.578	165.86	725.89	2752.2			
	CGS iterations	132	148	186	227	258			
	Memory/node (MB)	10.3	11.1	13.8	21.5	41.2			
$p = 8$	Jacobi time (s)	27.427	90.455	406.46	1777.1	6835.4			
	Jacobi iterations	1220	1500	1790	2095	2410			
	CGS time (s)	5.6017	26.682	106.98	458.67	1773.3	5850.3		
	CGS iterations	122	159	184	222	268	315		
	Memory/node (MB)	10.1	10.6	12.2	16.6	27.8	54.2		
$p = 12$	Jacobi time (s)	28.245	89.047	349.54	1590.6	5132.2	17187	44911	
	Jacobi iterations	1220	1500	1790	2095	2410	2730	3065	
	CGS time (s)	6.6556	27.423	91.973	351.43	1293.3	4379.83	23558	
	CGS iterations	126	166	184	217	257	322	320	
	Memory/node (MB)	10.1	10.5	11.6	14.9	23.4	43.2	86.2	
$p = 16$	Jacobi time (s)	31.880	95.864	348.39	1316.1	4664.8	14636	38770	
	Jacobi iterations	1220	1500	1790	2095	2410	2730	3065	
	CGS time (s)	7.1523	26.483	89.230	333.35	1183.2	3818.5	11058	62261
	CGS iterations	123	162	185	227	254	317	329	391
	Memory/node (MB)	10.1	10.4	11.3	14.1	21.1	37.6	73.5	92.0

Table 1: Real time in seconds required for the distributed solution of the FMS model.

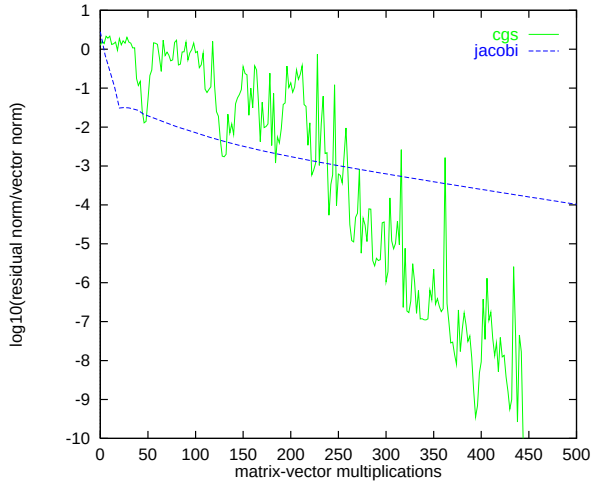


Figure 8: Jacobi and CGS convergence behaviour for the case $k = 7$.

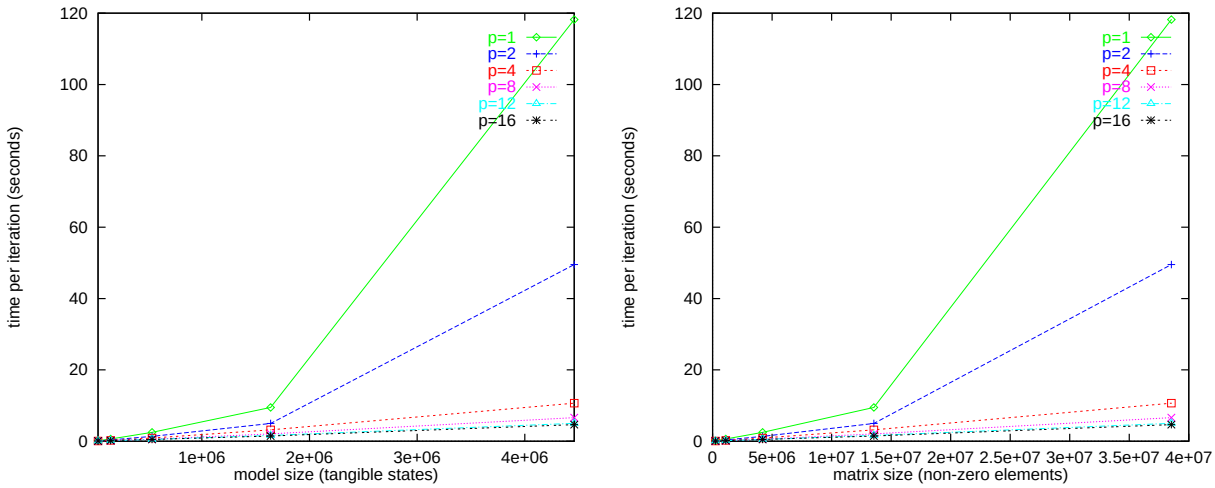


Figure 9: Time per distributed CGS iteration in seconds relative to the number of states (left) and number of transitions (right) in the model.

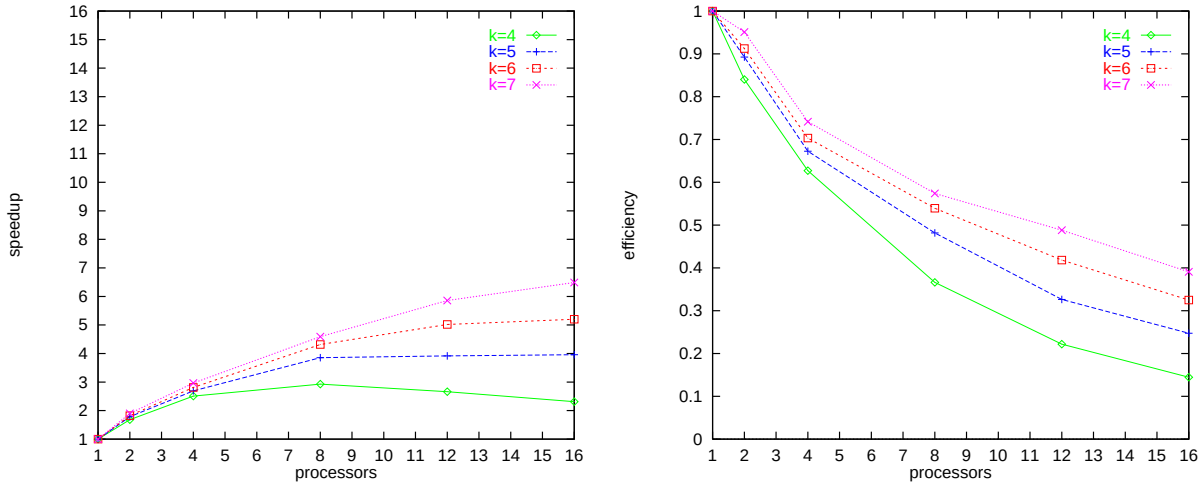


Figure 10: CGS speedup (left) and efficiency (right) for $k = 4, 5, 6, 7$. The case $k = 8$ gives superlinear speedup due to caching effects and is not shown.

and that adding processors increases the speedup for all but the smallest problems where communication costs dominate. The efficiency graph shows that diminishing returns occur as we add processors, as is to be expected on a sparse problem with low bandwidth.

Moving beyond the maximum problem size that can be generated on a single processor, the $k = 9$ case (11 million states) is solved in little over an hour on 16 processors, while the $k = 10$ case (25 million states) takes just over 3 hours. In the latter case, the total amount of disk I/O across all nodes is in excess of 1TB, with the nodes jointly processing an average of 91MB of disk data every second.

The largest case with $k = 11$ (54 million states) is solved in 17 hours 20 minutes on 16 processors. Here the total amount of disk I/O required is almost 3TB, with the nodes jointly processing an average of 42MB of disk data every second. The capability to solve a problem of this scale is impressive.

6 Conclusion and future work

We have considered distributed disk-based techiques for solving very large Markov chains using a distributed-memory parallel computer. We selected appropriate numerical methods and investigated the structure of infinitesimal generator matrices produced by sequential and distributed breadth-first state space generators. We exploited this structure to develop a matrix-vector multiply kernel which has low memory usage, low communication cost and good load balance. The kernel also provides opportunities for the overlapping of communication and computation.

We have described a software architecture for a distributed disk-based Markov solver. This software architecture uses two processes per node which allows for disk I/O to proceed concurrently with computation and communication. We have implemented such a solver on a 16-node distributed-memory parallel computer and used it to solve models with up to 50 million states and 500 million non-zero elements. Solving such a large problem using a sequential solver running on a single processor would be a daunting task indeed – besides the huge amount of computation required, the memory required to store the solution vector alone is over 400MB.

This study has concentrated on two scalable well-known numerical methods (Jacobi and Conjugate Gradient Squared). Future work will focus on developing distributed algorithms that also scale well but which use less memory and allow for the reuse of matrix blocks as they are generated.

7 Acknowledgements

The authors would like to thank the Imperial College Parallel Computing Centre for the use of the AP3000 distributed-memory parallel computer. We would also like to thank Silvana Zappacosta for helpful discussion. William Knottenbelt gratefully acknowledges the support and funding provided by the Beit Fellowship for Scientific Research.

References

[AH97] S.C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. *Lecture Notes in Computer Science*, 1253, 1997.

[Buc95] P. Buchholz. Hierarchical Markovian models: Symmetries and aggregation. *Performance Evaluation*, 22:93–110, 1995.

[CCM95] S. Caselli, G. Conte, and P. Marenzoni. Parallel state exploration for GSPN models. In *Lecture Notes in Computer Science 935: Proceedings of the 16th International Conference on the Application and Theory and Petri Nets*. Springer Verlag, Turin, Italy, June 1995.

[CGN98] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, Winter 1998.

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

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

[Fre93] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, March 1993.

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

[HLP95] Bruce Hendrickson, Robert Leland, and Steve Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *International Journal of High Speed Computing*, 7(1):73–88, 1995.

[ITS97] H. Ishihata, M. Takahashi, and H. Sato. Hardware of AP3000 scalar parallel server. *Fujitsu Scientific and Technical Journal*, 33(1):24–30, June 1997.

[Kem95] P. Kemper. Numerical analysis of superposed GSPNs. In *Proc. of the Sixth International Workshop on Petri Nets and Perfromance Models*, pages 52–62. IEEE Computer Society Press, 1995.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings Publishing, 1994.

[KK98] George Karypis and Vipin 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`.

[KMHK98] William J. Knottenbelt, Mark Mestern, Peter Harrison, and Pieter Kritzinger. Probability, parallelism and the state space exploration problem. In Ramon Puijaner, Nunzio N. Savino, and Bartomeu Serra, editors, *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.

[LG93] John G. Lewis and Robert A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Proceedings Supercomputing '93*, pages 484–492, Portland, Oregon, 15–19 November 1993. IEEE Computer Society Press.

[LPG94] John G. Lewis, David G. Payne, and Robert A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Scalable High Performance Computing Conference*, 1994.

[OA93] Andrew T. Ogielski and William Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, May 1993.

[Son89] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52, January 1989.

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

[Vor92] Henk van der 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*, 13(2):631–644, March 1992.

[Wei95] R. Weiss. A theoretical overview of Krylov subspace methods. *Appl. Numer. Math.*, 19:207–233, 1995. Special Issue on Iterative Methods for Linear Equations.