A Probabilistic Dynamic Technique for the Distributed Generation of Very Large State Spaces

W.J. Knottenbelt [a,1], M.A. Mestern [b], P.G. Harrison [a], P.S. Kritzinger [b]

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

[b] *Department of Computer Science, University of Cape Town, Rondebosch 7701, South Africa*

Abstract

Conventional methods for state space exploration are limited to the analysis of small systems because they suffer from excessive memory and computational requirements. We have developed a new dynamic probabilistic state exploration algorithm which addresses this problem for general, structurally unrestricted state spaces.

Our method has a low state omission probability and low memory usage that is independent of the length of the state vector. In addition, the algorithm can be easily parallelised. This combination of probability and parallelism enables us to rapidly explore state spaces that are an order of magnitude larger than those obtainable using conventional exhaustive techniques.

We derive a performance model of this new algorithm in order to quantify its benefits in terms of distributed run-time, speedup and efficiency. We implement our technique on a distributed-memory parallel computer and demonstrate results which compare favourably with the performance model. Finally, we discuss suitable choices for the three hash functions upon which our algorithm is based.

Keywords: state space exploration, state space generation, parallel algorithm, probabilistic algorithm

[1] William Knottenbelt is the Beit Fellow for Scientific Research in the Department of Computing at Imperial College. Author to use for correspondence. Email: wjk@doc.ic.ac.uk.

1 Introduction

Complex systems can be modelled using high-level formalisms such as stochastic Petri nets and process algebras. Often the first phase in the logical and numerical analysis of these systems is the explicit generation and storage of the model's underlying state space and state transition graph. In special cases, where the state space has sufficient structure, an efficient analytical solution can be obtained without the explicit enumeration of the entire state space. Several ingenious techniques, predominantly based on the theory of queueing networks, can be applied in such cases [3]. Further, certain restricted hierarchical structures allow states to be aggregated and the state space to be decomposed [5,16]. In this paper, however, we consider the general problem where no symmetry or other structure is assumed.

Conventional state space exploration techniques have high memory requirements and are very computationally intensive; they are thus unsuitable for generating the very large state spaces of real-world systems. Various authors have proposed ways of solving this problem by either using shared-memory multiprocessors [2] or by distributing the memory requirements over several computers in a network [7,6].

Allmaier *et al.* [2] present a parallel shared memory algorithm for the analysis of Generalised Stochastic Petri Nets (GSPNs) [1]. The shared memory approach means that there is no need to partition the state space as must be done in the case of distributed memory. This also brings the advantage of simplifying the load balancing problem. However, it does introduce synchronisation problems between the processors. Their technique is tested on a Convex SPP 1600 shared memory multiprocessor with 4GB of main memory. The authors observe good speedups for a range of numbers of processors employed and the system can handle 4 000 000 states with 2GB of memory.

Caselli *et al.* [6] offer two ways to parallelise the state space generation for massively parallel machines. In the data-parallel method, a marking of a GSPN with t transitions is assigned to t processors. Each processor handles the firing of one transition only and is responsible for determining the resulting state. This method was tested on a Connection Machine CM-5 and showed computation times linear in relation to the number of states. In the message-passing method the state space is partitioned between processors by a hash function and newly discovered states are passed to their respective processors. This method achieved good speedups on the CM-5, but was found to be subject to load imbalance.

Ciardo *et al.* [7] present an algorithm for state space exploration on a network of workstations. Their approach is not limited to GSPNs but has a general

interface for describing state transition systems. Their method partitions the state space in a way similar to [6] but they give no details of the storage techniques they use. The importance of a hashing function which evenly distributes the states across the processors is emphasised, but the method also attempts to reduce the number of states sent between processors. It was tested on a network of SPARC workstations interconnected by an Ethernet network and on an IBM SP-2 multiprocessor. In both cases a good reduction in processing time was reported although with larger numbers of processors, diminishing returns occurred. The largest state space successfully explored had 4 500 000 states; this required four hours of processing on a 32-node IBM SP-2.

None of the techniques proposed so far take advantage of the considerable gains achieved by using dynamic storage techniques based on hash compaction. The dynamic storage method we present here has several important advantages: memory consumption is low, space is not wasted by a static allocation and access to states is simple and rapid. We also present a parallel version of our technique which results in further performance gains.

After introducing the problem of state space exploration in Section 2, we give the details of the storage allocation algorithm in Section 3 and of the parallel state space generation algorithm in Section 4. A theoretical performance model is developed in Section 5 and numerical results demonstrating the observed performance of the algorithm are given in Section 6. Section 7 discusses suitable hashing and partition functions and Section 8 concludes and considers future work.

2 State Space Exploration

Fig. 1 shows an outline of a simple sequential state space exploration algorithm. The core of the algorithm performs a breadth-first search (BFS) traversal of a model's underlying state graph, starting from some initial state s_0. This requires two data structures: a FIFO queue F which is used to store unexplored states and a table of explored states E used to prevent redundant state exploration. The resulting breadth-first generation strategy is preferred over the alternative depth-first approach since it enables efficient row-by-row generation of the state graph A.

The function succ(s) returns the set of successor states of s. Some formalisms (such as GSPNs) include support for "instantaneous events" which occur in zero time. A state which enables an "instananeous event" is known as a *vanishing state*. We will assume that our successor function implements one of several known on-the-fly techniques available for eliminating vanishing states [8,17]. In addition, we will not consider the case where s_0 is vanishing.

```
begin
    E = {s_0}
    F.push(s_0)
    A = ∅
    while (F not empty) do begin
        F.pop(s)
        for each s' ∈ succ(s) do begin
            if s' ∉ E do begin
                F.push(s')
                E = E ∪ {s'}
            end
            A = A ∪ {id(s) → id(s')}
        end
    end
end
```

Fig. 1. Sequential state space generation algorithm

As the algorithm proceeds, it constructs A, the state graph. To save space, the states are identified by a unique state sequence number given by the function $\mathrm{id}(s)$. If we require the equilibrium state space probability distribution, we must construct a Markov chain by storing in A the transition rate between state s and s' for every arc $s \to s'$. The graph A is written out to disk as the algorithm proceeds, so there is no need to store it in main memory.

3 Dynamic Probabilistic Hash Table Compaction

The memory consumed by the state exploration process depends on the layout and management of the two main data structures of Fig. 1. The FIFO queue can grow to a considerable size in complex models. However, since it is accessed sequentially at either end, it is possible to manage the queue efficiently by storing the head and tail sections in main memory, with the central body of the queue stored on disk. The table of explored states, on the other hand, enjoys no such locality of access, and it has to be able to rapidly store and retrieve information about every reachable state. A good design for this structure is therefore crucial to the space and time efficiency of a state generator.

One way to manage the explored state table is to store the full state descriptor of every state in the state table. Such *exhaustive* techniques guarantee complete state coverage by uniquely identifying each state. However, the high memory requirements of this approach severely limit the number of states that can be stored. *Probabilistic* techniques, on the other hand, use hashing techniques to drastically reduce the memory required to store states. This reduction comes

at a cost, however, and it is possible that the hash table will represent two distinct states in the same way. If this should happen, the state hash table will incorrectly report a state as previously explored. This will result in incorrect transitions in the state graph and the omission of some states from the hash table. This risk may be acceptable if the probability of inadvertently omitting one or more states can be kept very small.

Probabilistic methods first gained widespread popularity with the development of Holzmann's bit-state hashing technique [13,14]. This technique aims at maximizing state coverage in the face of limited memory by using a hash function to map each state onto a single bit position in a large bit vector. Holzmann's method was subsequently improved upon by Wolper and Leroy's hash compaction technique [20], and Stern and Dill's enhanced hash compaction method [19]. These techniques hash states onto compressed values which are inserted into a large pre-allocated hash table with a fixed number of slots.

All of these probabilistic methods rely on *static* memory allocation, since they pre-allocate large blocks of memory for the explored-state table. Since the number of states in the system is in general not known beforehand, the pre-allocated memory may not be sufficient, or may be a gross overestimation. We now introduce a new probabilistic technique which uses *dynamic* storage allocation and which yields a very low collision avoidance probability.

The system is illustrated in Fig. 2. The explored state table takes the form of a hash table with several rows. Attached to each row is a linked list which stores compressed state descriptors. Two independent hash functions are used.

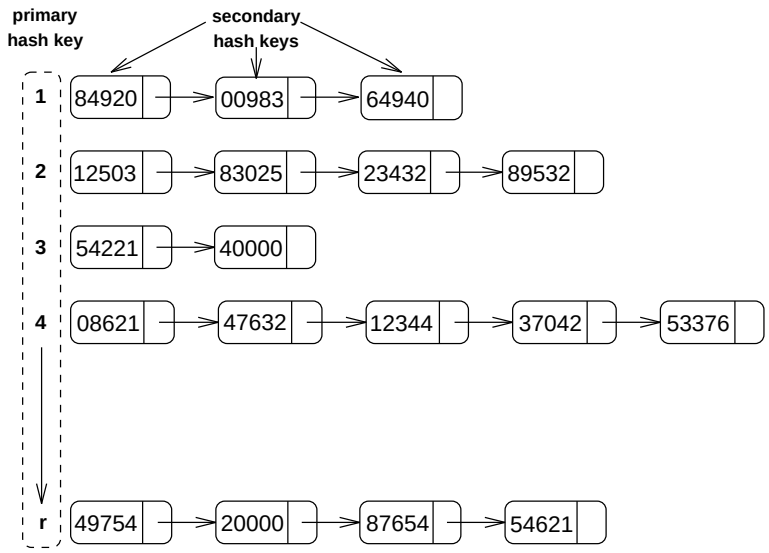

Fig. 2. Hash table with compressed state information

The *primary* hash function $h_1(s)$ is used to determine which hash table row should be used to store a compressed state and the *secondary* hash function

$h_2(s)$ is used to compute the compressed state descriptor values (also known as secondary keys). If a state's secondary key $h_2(s)$ is present in the hash table row given by its primary key $h_1(s)$, then the state is deemed to have been explored. Otherwise the secondary key is added to the hash table row and its successors are pushed onto the FIFO queue. Note that two states s_1 and s_2 are classified as being equal if and only if $h_1(s_1) = h_1(s_2)$ and $h_2(s_1) = h_2(s_2)$; this may happen even when the two state descriptors are different, so collisions may occur (as in all other probabilistic methods). As we will see in the next section, however, the probability of such a collision can be kept very small – certainly much smaller than the chance of a serious man-made error in the specification of the model. In addition, by regenerating the state space with a different independent set of hash functions and comparing the resulting number of states and transitions, it is possible to further arbitrarily decrease the risk of an undetected collision.

3.1 Reliability of the probabilistic dynamic state hash table

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

$$p = \frac{(rt)!}{(rt-n)!(rt)^n} \tag{1}$$

Using Stirling's approximation for $n!$ in Eq. (1) yields:

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

If $n^2 << rt$ (as will be the case in practical schemes with p close to 1), we can use the fact that $e^x \approx (1+x)$ for $|x| << 1$ to approximate p by:

$$p \approx 1 - \frac{n^2}{rt}$$

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

$$q = 1 - p \approx \frac{n^2}{rt} = \frac{n^2}{r2^b} \tag{2}$$

Thus the probability of state omission q is proportional to n^2 and is inversely proportional to the hash table size r. Increasing the size of the compressed

state descriptors b by one bit halves the omission probability.

3.2 Space complexity

If we assume that the hash table rows are implemented as dynamic arrays, the number of bytes of memory required by the scheme is:

$$M = hr + nb/8. \tag{3}$$

Here h is the number of bytes of overhead per hash table row. For a given number of states and a desired omission probability, there are a number of choices for r and b which all lead to schemes having different memory requirements. How can we choose r and b to minimize the amount of memory required? Rewriting Eq. (2):

$$r \approx \frac{n^2}{q2^b} \tag{4}$$

and substituting this into Eq. (3) yields

$$M \approx \frac{hn^2}{q2^b} + \frac{nb}{8}$$

Minimizing M with respect to b gives:

$$\frac{\partial M}{\partial b} \approx -\frac{n^2(\ln 2)h}{q2^b} + n/8 = 0$$

Solving for the optimal value of b at a specified state omission probability q yields:

$$b \approx \log_2\left(\frac{hn\ln 2}{q}\right) + 3$$

The corresponding optimal value of r can then be obtained by substituting b into Eq. (4).

Table 1 shows the the optimal memory requirements in megabytes (MB) and corresponding values of b and r for state space sizes ranging from 10^6 to 10^8. We have assumed a hash table row overhead of $h = 8$ bytes per row. In practice, it is difficult to implement schemes where b does not correspond to a whole number of bytes. Consequently, 4-byte or 5-byte compression is recommended.

q	number of states								
	10^6			10^7			10^8		
	MB	b	r	MB	b	r	MB	b	r
0.001	4.608	35	29104	50.21	39	181899	543.2	42	2273737
0.01	4.186	32	23283	46.08	35	291038	502.1	39	1818989
0.1	3.774	29	18626	41.86	32	232831	460.8	35	2910383

Table 1
Optimal values for memory usage and the values for b and r used to obtain them for various system state sizes and omission probabilities q

4 Parallel State Space Exploration

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

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

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

It is important that $h_0(s)$ achieves a good spread of states across nodes in order to achieve good load balance. Naturally, the values produced by $h_0(s)$ should also be independent of those produced by $h_1(s)$ and $h_2(s)$ to enhance the reliability of the algorithm.

The operation of node i in the parallel algorithm is shown in Fig. 3. Each node i has a local FIFO queue F_i used to hold unexplored local states and a hash table used to store the set E_i representing the states that have been explored locally. State s is assigned to processor $h_0(s)$, which stores the state's compressed state descriptor $h_2(s)$ in the local hash table row given by $h_1(s)$.

As in the sequential case, node i proceeds by popping a state off the local FIFO queue and determining the set of successor states. Successor states for

```
begin
    if h_0(s_0) = i do begin
        E_i = {s_0}
        F_i.push(s_0)
    end else
        E_i = {}
    A_i = ∅
    while (shutdown signal not received) do begin
        if (F_i not empty) do begin
            F_i.pop(s)
            for each s' ∈ succ(s) do begin
                if h_0(s') = i do begin
                    if s' ∉ E_i do begin
                        F_i.push(s')
                        E_i = E_i ∪ {s'}
                    end
                    A_i = A_i ∪ {id(s) → id(s')}
                end else
                    send-state(h_0(s'), id(s), s')
            end
        end
        while (receive-id(g, h)) do
            A_i = A_i ∪ {g→h}
        while (receive-state(k, g, s')) do begin
            if s' ∉ E_i do begin
                F_i.push(s')
                E_i = E_i ∪ {s'}
            end
            send-id(k, g, id(s'))
        end
    end
end
```

Fig. 3. Parallel state space generation algorithm for node i

which $h_0(s) = i$ are dealt with locally, while other successor states are sent to the relevant remote processors via calls to send-state(k, g, s). Here k is the remote node, g is the identity of the parent state and s is the state descriptor of the child state. The remote processors must receive incoming states via matching calls to receive-state(k, g, s) where k is the sender node. If they are not already present, the remote processor adds the incoming states to both the remote state hash table and FIFO queue.

For the purpose of constructing the state graph, states are identified by a pair of integers (i, j) where $i = h_0(s)$ is the node number of the host processor

and j is the local state sequence number. As in the sequential case, the index j can be stored in the state hash table of node i. However, a node will not be aware of the state identity numbers of non-local successor states. When a node receives a state it returns its identity to the sender by calling send-id(k, g, h) where k is the sender, g is the identity of the parent state and h is the identity of the received state. The identity is received by the original sender via a call to receive-id(g, h).

In practice, it is inefficient to implement the communication as detailed in Fig. 3, since the network rapidly becomes overloaded with too many short messages. Consequently state and identity messages are buffered and sent in large blocks. In order to avoid starvation and deadlock, nodes that have very few states left in their FIFO queue or are idle broadcast a message to other nodes requesting them to flush their outgoing message buffers.

The algorithm terminates when all the F_i's are empty and there are no outstanding state or identity messages. We use Dijkstra's circulating probe algorithm [10] to determine when this occurs.

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

$$q = \frac{n^2}{Nr2^b} \tag{5}$$

5 A Theoretical Performance Model

We now develop a model for the predicting the run-time and speedup of our algorithm when implemented on a statically-routed wraparound mesh of N processors. The model is based on the calculation of two key quantities: the *computation time* $T_W(N)$ required to generate arcs and search for states in the local hash table, and the *communication time* $T_C(N)$ required to send and receive non-local states. Predicted run-time is then simply given by $T_W(N) + T_C(N)$.

For the purposes of this analysis, we ignore the start-up period and termination phase and we assume that the FIFO queue is never empty in any processor. These are reasonable assumptions for problems with large state spaces – certainly for any algorithm that runs for more than a few minutes. Further, the randomness in the hash functions is assumed to achieve perfect load balanc-

ing so that, after the start-up period and before the termination phase of the algorithm, all processors operate functionally in the same way as per Fig. 3.

We assume that a total of a arcs are generated in total and that there are a total of n unique states (nodes) in the state graph. A processor takes c seconds to construct the destination state corresponding to an arc in the state graph. Further, each local arc requires a search to be performed on a row in the local hash table. Each processor's hash table has r rows and it takes an average of d seconds to scan each entry in a row. Note that the value of c is likely to vary between models (depending on such factors as the proportion of vanishing states in the state graph), while the value of d remains constant between models.

Assuming ideal random hash functions which distribute states and arcs evenly over processors, each processor will generate a/N (mostly non-local) arcs and will process a/N local arcs. Each state hash table row on each processor will contain an average of $n/(2Nr)$ elements over the lifetime of the hash table. The computation time T_W as a function of the number of processors N is thus estimated by:

$$T_W(N) = \frac{a}{N}(c + \frac{dn}{2Nr}).$$

The number of non-local arcs m generated per processor, assuming that new destination states generated belong to each of the N processors with equal probability, is simply

$$m = \frac{a}{N}\frac{N-1}{N} = \frac{a(N-1)}{N^2}$$

The processing of a non-local arc is assumed to generate L bytes of data traffic. To prevent the communication network from being overwhelmed by thousands of short messages, state and identity messages for non-local arcs are buffered and sent in blocks between processors. The overhead associated with buffer management for each arc is assumed to be s seconds. We assume that buffers are transmitted over the network when they become full with B bytes of data, using a blocking I/O cut-through transfer. Messages are divided into flits of size F bytes, which are sent between adjacent nodes serially. The header of any message consists of one flit and acknowledgement messages comprise only a header flit. The per-hop latency for one flit is f seconds.

The mean path length for a square wraparound mesh of N processors (for N the square of an even number) is $1 + \sqrt{N}/2$ [2] and the total number of buffers sent (and also their acknowledgements) is mL/B.

[2] In fact, the mesh we use in the results section is rectangular 2×6 and so the mean path length is $1 + 2 = 3$ which is close to our estimate of $1 + \sqrt{N}/2$ for $N = 12$.

Mean header transfer time is therefore $(1+\sqrt{N}/2)f$ and message transmission time is Bf/F. Thus for data, average communication time is

$$m[s + \frac{L}{B}(1 + \sqrt{N}/2 + \frac{B}{F})f]$$

and for an acknowledgement,

$$\frac{mL}{B}(1 + \sqrt{N}/2)f$$

Hence the total time T_C spent by each processor on communication overhead is, on average,

$$T_C(N) = m[s + \frac{2L}{B}(1 + \sqrt{N}/2 + \frac{B}{2F})f]$$

The speed-up of the algorithm executing on this architecture can now be calculated as:

$$S(N) = T_W(1)/(T_W(N) + T_C(N))$$

and its efficiency is given by $E(N) = S(N)/N$.

Notice that the algorithm is not cost-optimal because its cost (the product of the parallel run time and the number of processors used) is given by

$$C(N) = N(T_W(N) + T_C(N))$$

which cannot be proportional to $T_W(1)$ for large N, on account of the $\sqrt{N}/2$ term in $T_C(N)$. Since it is impossible to maintain the efficiency at a constant value by simply increasing the size of the state graph, the algorithm is technically not scalable for very large N. However, since $\sqrt{N}/2$ grows slowly and is typically negligible in comparison with $B/2F$ for moderate N, the algorithm's efficiency is maintained well for machines with up to a few hundred processors.

6 Results

To illustrate the potential of our technique, we consider the 22-place GSPN model of a flexible manufacturing system shown in Fig. 4. This model, which we will refer to as the FMS model, was originally presented in detail in [9], and was subsequently used in [7] to demonstrate distributed exhaustive state space generation. A detailed understanding of the model is not required. It suffices to note that the model has a parameter k (corresponding to the number of initial tokens in places $P1, P2$ and $P3$), and that as k increases, so does the number of states n and the number of arcs a in the state graph (see Fig. 5).

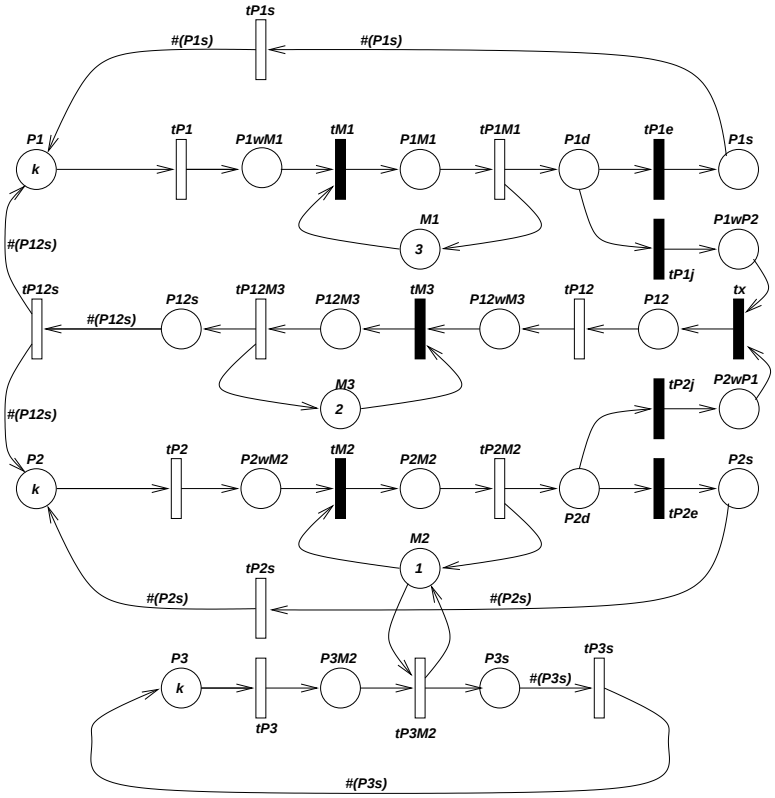

Fig. 4. The FMS Generalised Stochastic Petri net [9]

We implemented the state generator algorithm of Fig. 3 using hash tables with $r = 350\,003$ rows per processor and $b = 40$ bit secondary keys. The generator was written in C++, with support for two popular parallel programming interfaces, viz. the Message Passing Interface (MPI) [12] and the Parallel Virtual Machine (PVM) interface [11]. Models are specified using the DNAmaca interface language [17] which allows the high-level specification of generalised timed transition systems including GSPNs, queueing networks and Queueing Petri nets [4]. The high-level specification is then translated into a C++ class which is compiled and linked to a library implementing the core state generator. The state space and state graph are written to disk in compressed format as the algorithm proceeds.

We obtained our results on a Fujitsu AP3000 distributed-memory parallel computer with 12 processing nodes [15]. Each node has a 200 MHz UltraSparc processor, 256MB RAM and 4GB local disk space. The nodes run the Solaris operating system and support MPI. They are connected by a high-speed wormhole-routed network with a peak throughput of 200MB/s (the AP-net).

k	n	a
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
12	111 414 940	1 078 917 632

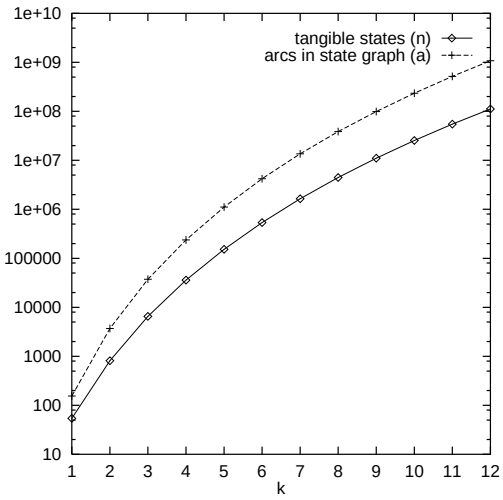

Fig. 5. The number of tangible states (n) and the number of arcs in the state graph (a) for various values of k

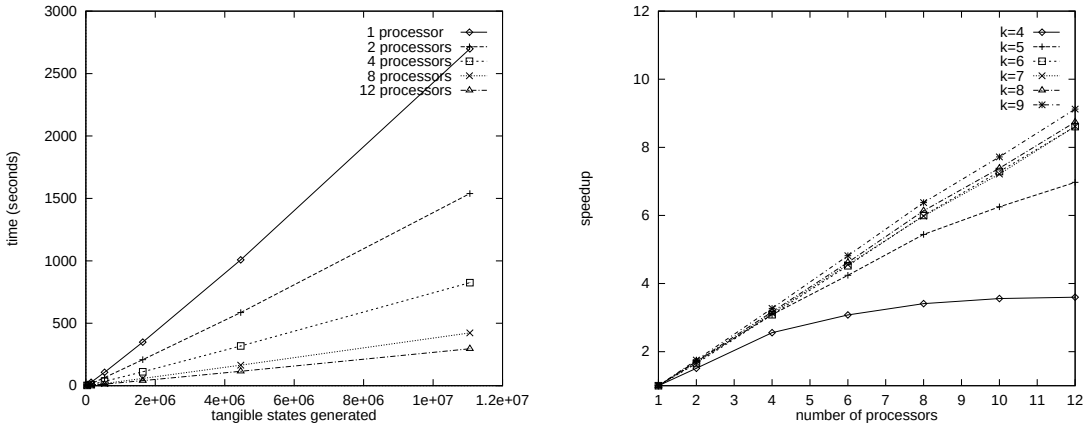

Fig. 6. Real time taken to generate state spaces up to $k = 9$ using 1, 2, 4, 8 and 12 processors (left), and the resulting speedups for $k = 4, 5, 6, 7, 8$ and 9 (right)

6.1 Run-times and speedup

The graph on the left in Fig. 6 shows the time (defined as the maximum processor run-time) taken to explore state spaces of different sizes (up to $k = 9$) using 1, 2, 4, 8 and 12 processors on the AP3000. Each observed run-time value is calculated as the mean run-time of four runs on the AP3000. The $k = 8$ state space (4 459 455 states) can be generated on a single processor in

under 17 minutes; 12 processors require just 115 seconds. The $k = 9$ state space (11 058 190 states) can be generated on a single processor in 45 minutes; 12 processors require just 296 seconds.

The graph on the right in Fig. 6 shows the speedups for the cases $k = 4, 5, 6, 7, 8, 9$. The speedup for N processors is given by the run time of the sequential generation ($N = 1$) divided by the run time of the distributed generation with N processors. For $k = 9$ using 12 processors we observed a speedup of 9.12, giving an efficiency of 76%. Most of the lost efficiency can be accounted for by communication overhead and buffer management, which is not present in the sequential case. Since speedup increases linearly in the number of processors for $k > 6$, there is evidence to suggest that our algorithm scales well.

The memory utilization of our technique is low: a single processor generating the $k = 8$ state space uses a total of 74MB RAM (16.6 bytes per state), while the $k = 9$ state space requires 160MB RAM (14.5 bytes per state). 9 bytes of the memory used per state can be accounted for by the 40-bit secondary key and the 32-bit unique state identifier; the remainder can be attributed to factors such as hash table overhead and storage for the front and back of the unexplored state queue. By comparison, a minimum of 48 bytes would be required to store a state descriptor in a straightforward exhaustive implementation (22 16-bit integers plus a 32-bit unique state identifier). The difference will be even more marked with more complex models that have longer state descriptors, since the memory consumption of our technique is independent of the number of elements in the state descriptor.

6.2 Larger state graphs

Moving beyond the maximum state space size that can be generated on a single processor, the graph on the left in Fig. 7 shows the real time required to generate larger state spaces using 12 processors. For the largest case ($k = 12$) 55 minutes are required to generate a state space with 111 414 940 tangible states and a state graph with 1 078 917 632 arcs. The graph on the right in Fig. 7 shows the distribution of the states generated by each processor for the case $k = 12$.

In comparison to the results reported above (see Table 5), Ciardo *et al* used conventional exhaustive distributed generation techniques to generate the same sample model for the case $k = 8$ in 4 hours using 32 processors on an IBM SP-2 parallel computer [7]. They were unable to explore state spaces for larger values of k.

To enhance our confidence in our results for the case $k = 12$, we use Eq. (5)

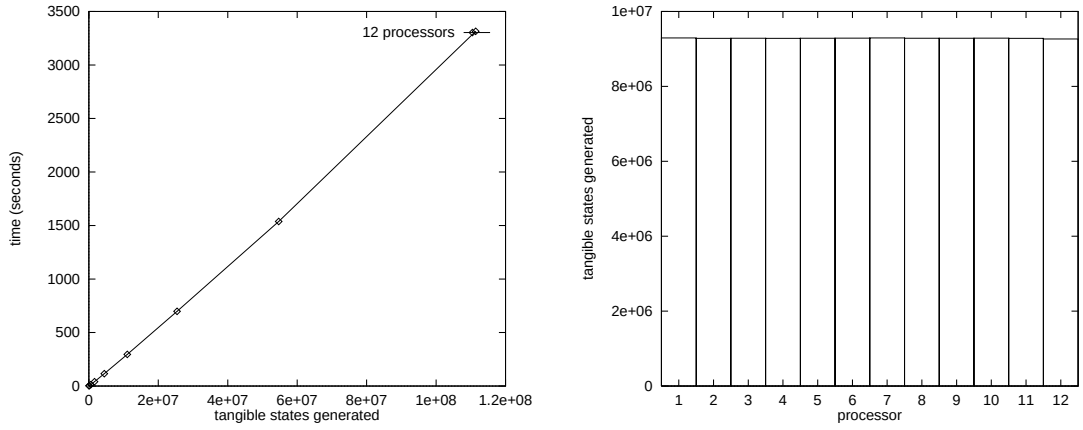

Fig. 7. Real time taken to generate state spaces up to $k = 12$ using 12 processors (left) and distribution of states across processors for $k = 12$ (right)

to compute the probability of having omitted at least one state. For a state space of size $n = 10^8$ states, the omission probability q is given by:

$$q \approx \frac{n^2}{Nr2^b} = \frac{10^{16}}{12 * 350\,003 * 2^{40}} = 0.00217$$

i.e. the omission probability is approximately 0.2%. This is a small price to pay for the ability to explore such large state spaces, and is probably less than the chance of a serious (man-made) error in specifying the model.

To further increase our confidence in the results, we changed all three hash functions and regenerated the state space. This resulted in exactly the same number of tangible states and arcs. This process could be repeated several times to establish an even higher level of confidence in the results.

6.3 Validation of the performance model

We assess the accuracy of the performance model presented in Section 5 by comparing observed results with model predictions for the FMS model running on the AP3000. The values used to parameterise our performance model are given in Fig. 8. Agreement of observed and predicted run-times is excellent, as shown in the graphs of Fig. 9, and Appendix B which gives the full data set of observed and predicted values together with the relative model error expressed as a percentage.

In the single processor case, which does not involve any communication, predicted run-times are typically well within 1% of the observed values, suggesting that our model for T_W is very accurate. Predicted run-times for multiple processor runs involving communication are typically within 5% of the observed values, with a tendency for the model to predict a slightly lower run-time

	Parameter description	Value
N	number of nodes	(variable)
a	number of arcs in state graph	(variable)
n	number of states in state space	(variable)
F	flit size	32 bits
L	comms induced by one non-local arc	60 bytes
B	message buffer size	8192 bytes
f	per-hop flit latency	220 ns
c	cost of generating one arc	25.4 μs
d	cost of scanning one hash table entry	120 ns
s	cost of buffer management for non-local arc	6.50 μs
r	number of hash table rows per node	350 003 rows

Fig. 8. Parameters used in the performance model for the FMS model running on the AP3000

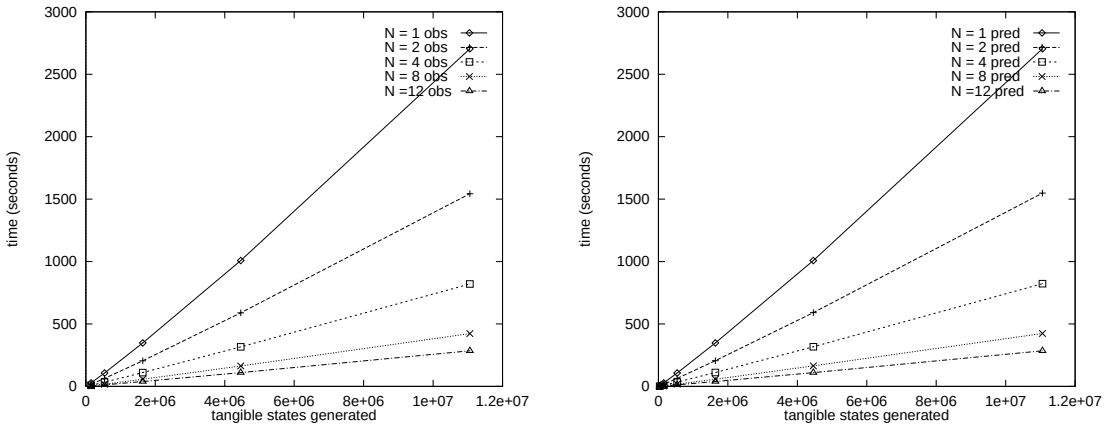

Fig. 9. Observed (left) and predicted (right) real time taken to generate state spaces up to $k = 9$ using 1, 2, 4, 8 and 12 processors

than that which is observed. This is not surprising since our model is based on ideal assumptions such as hash functions which achieve perfect load balancing of communication load. In addition, in those cases where there are a small number of states per processor, the start-up and termination phase requires a significant proportion of the run-time, and this is not accounted for by the model.

There is also good agreement between the observed and predicted speedup values, as shown in Fig. 10 and Appendix B. For the reasons outlined in the previous paragraph, there is a tendency for the model to predict a slightly

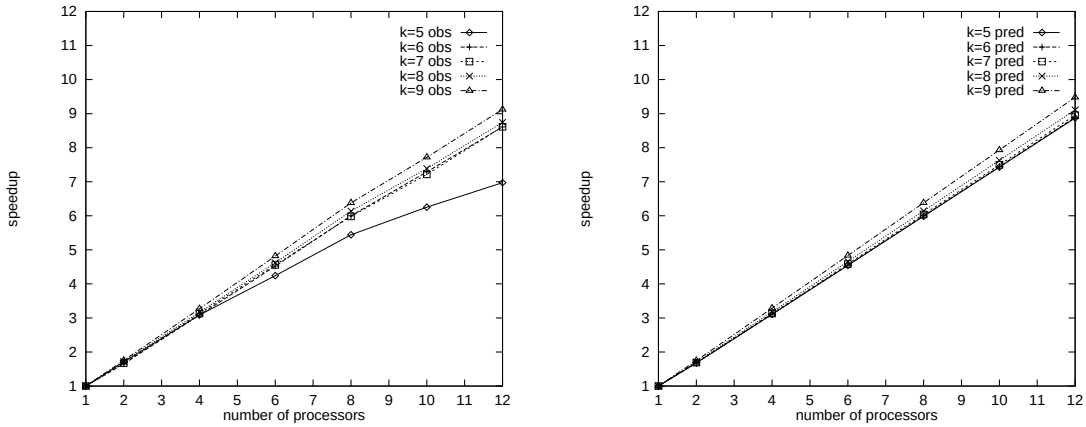

Fig. 10. Observed (left) and predicted (right) speedups for $k = 5, 6, 7, 8$ and 9

higher speedup than that which is actually observed.

7 Choosing good hash functions

Recall that our technique is based on the use of the following three hash functions:

- the **partitioning hash function** $h_0(s) \rightarrow \{0, 1, \ldots, N-1\}$, which assigns state s to a processor.
- the **primary hash function** $h_1(s) \rightarrow \{0, 1, \ldots, r-1\}$ which assigns state s to a row in the hash table on processor $h_0(s)$.
- the **secondary hash function** $h_2(s) \rightarrow \{0, 1, \ldots, 2^b - 1\}$ which maps state s onto a b-bit compressed value; this compressed value is stored in row $h_1(s)$ of the hash table on processor $h_0(s)$.

The reliability of our technique depends on the behaviour of these hash functions in three important ways. Firstly, h_0 and h_1 should randomly partition states across the processors and hash table rows. Secondly, h_2 should result in a random distribution of compressed values. Finally, h_0, h_1 and h_2 should distribute states independently of one other.

Before we consider each of these functions individually, consider the two general hash functions f_1 and f_2 shown in Fig. 11. Both map an m-element state vector $s = (s_1, s_2, \ldots, s_m)$ onto a 32-bit unsigned integer by manipulating the bit representations of individual state vector elements. The **xor** operator is the bitwise exclusive or operator, **rol** is the bitwise rotate-left operator and **mod** is the modulo (remainder) operator.

Hash function $f_1(s, \mathit{shift})$ uses exclusive or to combine rotated bit representations of the state vector elements. State vector element s_i is rotated left by

```
f1(vector s, int shift) → uint32
begin
  uint32 key = 0;
  int slide = 0;
  for i=1 to m do begin
    key = key xor (s_i rol slide);
    slide = (slide + shift) mod 32;
  end
  return key;
end
```

```
f2(vector s, int shift1 , int shift2) → uint32
begin
  uint32 key = 0;
  int slide1 = 0, slide2 = 16, sum = 0;
  for i=1 to m do begin
    sum = sum + s_i
    key = key xor (s_i rol slide1);
    key = key xor (sum rol slide2);
    slide1 = (slide1 + shift1) mod 32;
    slide2 = (slide2 + shift2) mod 32;
  end
  return key;
end
```

Fig. 11. Two general hash functions for mapping states onto 32 bit unsigned integers.

an offset of $(i \times \mathit{shift}) \bmod 32$ bits. Hash function $f_2(s, \mathit{shift}_1, \mathit{shift}_2)$ is based on encoding not only element s_i rotated left by an offset of $i \times \mathit{shift}_1 \bmod 32$, but also the sum $\sum_{j<i} s_i$ rotated left by an offset of $i \times \mathit{shift}_2 \bmod 32$. This technique makes the hash function resistant to any symmetries and invariants that may be present in the model.

We make use of functions f_1 and f_2 to derive suitable choices for $h_0(s)$, $h_1(s)$ and $h_2(s)$ as follows:

- For the **partitioning hash function**, we use either

  $$h_0(s) = f_1(s, \mathit{shift}) \bmod \mathit{prime} \bmod N$$

  or

  $$h_0(s) = f_2(s, \mathit{shift}_1, \mathit{shift}_2) \bmod \mathit{prime} \bmod N$$

  where *shift*, shift_1 and shift_2 are arbitrary shifting factors relatively prime to 32 and *prime* is some prime number $>> N$.
- For the **primary hash function**, we use either

  $$h_1(s) = f_1(s, \mathit{shift}) \bmod r$$

  or

  $$h_1(s) = f_2(s, \mathit{shift}_1, \mathit{shift}_2) \bmod r$$

  where *shift*, shift_1 and shift_2 are arbitrary shifting factors relatively prime to 32 and r, the number of rows in the hash table, is a prime number.
- For the **secondary hash function**, we consider 32-bit (4-byte) compression

based on either f_1 or f_2:

$$h_2(s) = f_1(s, shift)$$

or

$$h_2(s) = f_2(s, shift_1, shift_2)$$

where $shift$, $shift_1$ and $shift_2$ are relatively prime to 32. Function f_2 has the desirable property that it is resistant to symmetries and invariants in the model; this prevents similar (but distinct) states from having the same secondary hash values. Consequently, f_2 gives a better spread of secondary values then f_1. For 40-bit secondary hash keys (i.e. five-byte state compression), f_2 can easily be modified to produce a 40-bit hash key instead of a 32-bit hash key.

It is important to ensure the independence of the values produced by $h_0(s)$, $h_1(s)$ and $h_2(s)$. The following guidelines assist this:

- Some hash functions should be based on f_1 while others are based on f_2; hash functions which use the same base function should use different shifting factors.
- The hash functions should consider state vector elements in a different order.
- the value of r used by $h_1(s)$ should not be the same as the value of $prime$ used by $h_0(s)$.

The results presented in Section 6 made use of partitioning and primary functions based on f_1 and a 40-bit secondary hash function based on f_2. Appendix A presents graphs and tables illustrating the performance of these hash functions for the FMS model.

8 Conclusion and future work

We have presented a new dynamic probabilistic state exploration technique and developed an efficient parallel implementation that exhibits good scalability. In contrast to conventional state exploration algorithms, the memory usage of our technique is very low and is independent of the length of the state vector. Since the method is probabilistic, there is a chance of state omission, but the reliability of our technique is excellent and the probability of omitting one or more states is extremely small. Moreover, by performing multiple runs with independent sets of hash functions, we can reduce the omission probability almost arbitrarily at linear computational cost.

Our results to date show good speedups and scalability. It is the combina-

tion of probability and parallelism that dramatically reduces both the space and time requirements of large-scale state space exploration. We note here that the same algorithm could also be effectively implemented on a shared-memory multiprocessor architecture, using a single shared hash table and a shared breadth first search queue. There would be no need for a partitioning function and contention for rows in the shared hash table would be very small. Consequently, it should again be possible to achieve good speedups and scalability.

Our technique is based on the use of hashing functions to assign states to processors, hash table rows, and compressed state values. The reliability analysis requires that the hash functions distribute states randomly and independently and we have shown how to generate hashing functions which meet these requirements. To illustrate its potential, we have explored a state space with more than 10^8 tangible states and 10^9 arcs in under an hour using 12 processors on an AP3000 parallel computer. The probability of state omission is just 0.2%.

Previously, the memory and time bottleneck in the performance analysis pipeline has been state space exploration. We believe that our technique shifts this bottleneck away from state space generation and onto stages later in the analysis pipeline. Future work will therefore focus on a parallel functional analyser and a parallel steady-state solver. The functional analyser will ensure that the generated state graph maps onto an irreducible Markov chain by eliminating transient states and by verifying that the remaining states are strongly connected. The steady-state solver will then solve the state graph's underlying Markov chain for its steady-state probability distribution by using state-of-the-art linear equation solvers designed to cope with the large problem size. Indeed, recent experiments with distributed disk-based solution techniques have demonstrated the ability to solve for the steady-state distribution of very large models with over 50 million states and over 500 million transitions [18].

9 Acknowledgements

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

References

[1] M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.

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

[3] F. Basket, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, closed and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22:248 – 260, 1975.

[4] F. Bause. Queueing Petri nets: A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri nets and Performance Models*. IEEE, October 1993.

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

[6] 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.

[7] 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.

[8] G. Ciardo, J.K. Muppula, and K.S. Trivedi. On the solution of GSPN reward models. *Performance Evaluation*, 12(4):237–253, 1991.

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

[10] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing letters*, 16:217–219, June 1983.

[11] 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.

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

[13] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[14] G.J. Holzmann. An analysis of bitstate hashing. In *Proceedings of IFIP/PSTV95: Conference on Protocol Specification, Testing and Verification*. Chapman & Hall, Warsaw, Poland, June 1995.

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

[16] 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.

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

[18] 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)*, Zaragoza, Spain, 6–10 September 1999. To appear.

[19] U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.

[20] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Lecture Notes in Computer Science 697*, pages 59–70. Springer-Verlag, 1993.

A Appendix: Hash Function Performance

In this appendix we give detailed results showing how well the hash functions proposed in Section 7 meet their objectives of achieving a good spread of states over processors, hash table rows and secondary key values. We also evaluate the independence of the values produced by these hash functions.

A.1 Partitioning hash function

The graphs in Fig. A.1 show the distribution of state assignments in the FMS model with $k = 9$ and $N = 12$ for two partitioning hashing functions, one based on f_1, the other on f_2. Table A.1 compares the performance of these two hash functions against that of an ideal random hash function over a wider range of k and N values. The performance is expressed in terms of σ_N, the standard deviation of the number of states assigned to each processor, and we assume that an ideal random hash function distributes n states over N processors such that the number of states assigned to a processor follows a binomial distribution with parameters $(n, 1/N)$.

Both variants of the partitioning function give well-balanced state distributions. However, the function based on f_1 is preferable, since f_1 involves less

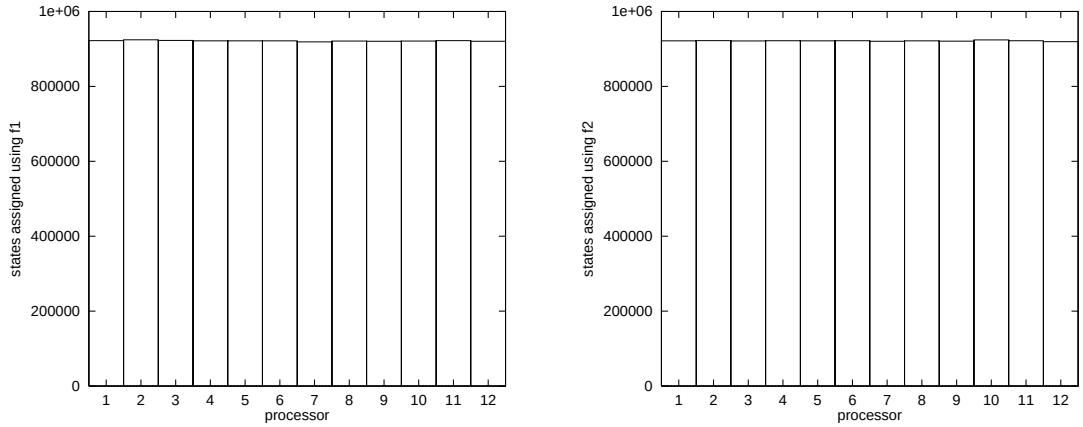

Fig. A.1. State distributions for the FMS model with $k = 9$ and $N = 12$ using $h_0(s) = f_1(s, 3) \bmod 5\,003 \bmod N$ (left) and $h_0(s) = f_2(s, 3, 5) \bmod 5\,003 \bmod N$ (right).

		σ_N								
	tangible	$N = 8$			$N = 12$			$N = 16$		
k	states	rnd	f_1	f_2	rnd	f_1	f_2	rnd	f_1	f_2
5	152 712	129	77	78	108	164	89	95	48	99
6	537 768	243	127	164	203	282	142	178	124	234
7	1 639 440	423	370	355	354	323	263	310	258	270
8	4 459 445	698	482	716	584	417	314	511	545	533
9	11 058 190	1 100	1 418	1 942	919	1 353	1 118	805	1 089	1 345

Table A.1
Values of σ_N, the standard deviation of the number of states allocated to each processor, for the FMS model using three partitioning functions: an ideal random hashing function, $h_0(s) = f_1(s, 3) \bmod 5\,003 \bmod N$ and $h_0(s) = f_2(s, 3, 5) \bmod 5\,003 \bmod N$.

computation than f_2. The even distribution of states ensures good load balancing of computation and communication overhead across processors, and also maintains the reliability of our technique.

A.2 Primary hash function

Table A.2 compares the performance of two primary hash functions against that of an ideal random hash function for the FMS model. We assume all states are inserted into a single hash table with $r = 350\,003$ rows. We express the performance of the hash functions in terms of the number of hash table rows used and in terms of σ_r^2, the variance of the hash table row length. We assume

	tangible	hash table rows used			σ_r^2		
k	states	random	f_1	f_2	random	f_1	f_2
5	152 712	123 757	122 349	123 809	0.436	0.448	0.436
6	537 768	274 704	271 493	274 611	1.536	1.618	1.542
7	1 639 440	346 769	345 932	346 743	4.684	5.165	4.672
8	4 459 445	350 002	350 001	350 001	12.741	14.670	12.741
9	11 058 190	350 003	350 003	350 003	31.595	39.391	31.694

Table A.2
Values of σ_r^2, the variance of the number of states allocated to each hash table row, and the number of hash table rows used for the FMS model. We take $r = 350\,003$ and consider three primary hash functions: an ideal random hashing function, $h_1(s) = f_1(s, 7) \bmod r$ and $h_1(s) = f_2(s, 3, 5)$.

that an ideal random hash function distributes n states over r rows such that the number of states assigned to each row follows a binomial distribution with parameters $(n, 1/r)$.

Both functions provide a good spread of states across hash table rows. If maximum computational speed is desirable the hash function based on f_1 provides a reasonable distribution of states. However, the hash function based on f_2 consistently achieves a better spread of states, so the hash function based on f_2 is better if maximum reliability is the main concern.

A.3 Secondary hash function

The reliability analysis of our technique requires that the secondary hash function achieves a good distribution of states over the possible key values. In addition, the values produced by $h_2(s)$ should be independent of the values produced by $h_0(s)$ and $h_1(s)$. This can be ensured using the techniques outlined in Section 7.

Table A.3 compares the performance of secondary hash functions $h_2(s) = f_1(s, 7)$ and $h_2(s) = f_2(s, 3, 5)$ with that of an ideal random hashing function for the states in the FMS model. The performance is expressed in terms of the number of unique secondary key values across all states. As before, we assume that an ideal random hash function distributes n states over 2^{32} possible key values such that the number of states assigned to each key value follows a binomial distribution with parameters $(n, 1/2^{32})$.

Hash function f_1 does not achieve a particularly good distribution of secondary key values, while f_2 consistently achieves an excellent state distribution better

	tangible	unique secondary key values		
k	states	random	f_1	f_2
5	152 712	152 707	149 694	152 712
6	537 768	537 701	519 530	537 730
7	1 639 440	1 638 814	1 540 241	1 639 058
8	4 459 455	4 454 827	4 063 882	4 456 835
9	11 058 190	11 029 755	9 544 696	11 043 283

Table A.3
The number of unique secondary key values obtained by applying three secondary hash functions to the states of the FMS model. We consider an ideal random hash function, $h_2(s) = f_1(s, 7)$ and $h_2(s) = f_2(s, 3, 5)$.

than the ideal random hash function.

A.4 Evaluating hash function independence

Table A.4 shows the correlation between hash function values for various values of k for the states of the FMS model. The results are presented in terms of r_{ij}, the correlation between the values produced by hash functions $h_i(s)$ and $h_j(s)$. None of the correlations is significantly different from zero (assuming a significance level of $\alpha = 0.05$ in Pearson's test for significant correlation).

	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$
r_{01}	4.35×10^{-3}	1.17×10^{-3}	4.38×10^{-4}	3.63×10^{-4}	-5.80×10^{-5}
r_{02}	2.74×10^{-3}	5.93×10^{-4}	-2.12×10^{-5}	8.56×10^{-5}	-2.37×10^{-4}
r_{12}	1.30×10^{-3}	4.28×10^{-3}	-1.66×10^{-4}	-1.41×10^{-4}	-7.87×10^{-5}

Table A.4
Correlations between hash function values for the states of the FMS model with $k = 4, 5, 6, 7, 8$. Here $N = 256$, $r = 350\,003$ and $b = 32$. The hash functions used are $h_0(s) = f_1(s, 3) \bmod 5003 \bmod N$, $h_1(s) = f_1(s, 7) \bmod r$ and $h_2(s) = f_2(s, 3, 5)$.

Fig. A.2 shows scattergrams of the hash function values of 10 000 states sampled from the state space of the FMS model with $k = 7$. No unusual clusters or patterns are observed in any of the scattergrams. The assumption that our hash functions distribute states independently of one another therefore seems to be reasonable in the context of the FMS model.

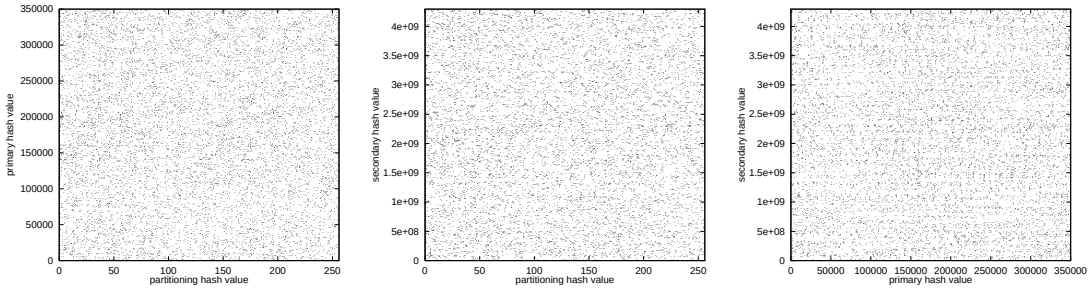

Fig. A.2. Scattergrams of the hash function values of a sample of 10 000 states taken from the state space of the FMS model with $k = 7$. The parameter values and hash functions are the same as those used in Table A.4.

B Appendix: Accuracy of the Performance Model

k	N	pred. run-time	obs. run-time	err. %	pred. speedup	obs. speedup
5	1	28.26	27.32	3.44	1.00	1.00
	2	16.85	16.00	5.31	1.67	1.70
	4	9.103	8.850	2.86	3.10	3.08
	6	6.220	6.440	-3.42	4.54	4.24
	8	4.722	5.018	-5.90	5.99	5.44
	10	3.805	4.368	-12.9	7.43	6.25
	12	3.186	3.918	-18.7	8.87	6.97
6	1	107.21	107.01	0.19	1.00	1.00
	2	63.82	64.80	-1.51	1.68	1.65
	4	34.46	34.70	-0.69	3.11	3.08
	6	23.54	23.67	-0.55	4.55	4.52
	8	17.87	17.84	0.16	6.00	6.00
	10	14.40	14.68	-1.91	7.45	7.29
	12	12.06	12.43	-2.98	8.89	8.61
7	1	348.05	349.07	-0.29	1.00	1.00
	2	206.30	208.64	-1.12	1.69	1.67
	4	111.22	111.54	-0.29	3.13	3.13
	6	75.94	76.64	-0.91	4.58	4.55
	8	57.63	58.35	-1.23	6.04	5.98

k	N	pred. run-time	obs. run-time	err. %	pred. speedup	obs. speedup
7	10	46.43	48.38	-4.03	7.50	7.22
	12	38.87	40.52	-4.07	8.96	8.61
8	1	1008.22	1008.42	-0.02	1.00	1.00
	2	591.21	586.24	0.85	1.71	1.72
	4	317.39	318.15	-0.24	3.18	3.17
	6	216.43	218.26	-0.84	4.66	4.62
	8	164.14	164.30	-0.10	6.14	6.14
	10	132.19	136.55	-3.19	7.63	7.39
	12	110.64	115.33	-4.07	9.11	8.74
9	1	2704.33	2698.01	0.23	1.00	1.00
	2	1548.08	1538.99	0.59	1.75	1.75
	4	823.04	825.03	-0.24	3.29	3.27
	6	559.59	559.99	-0.07	4.83	4.81
	8	423.78	423.08	0.17	6.38	6.37
	10	340.99	349.52	-2.44	7.93	7.72
	12	285.25	295.68	-3.53	9.48	9.12
10	12	679.22	697.88	-2.67	-	-
11	12	1518.37	1537.7	-1.26	-	-
12	12	3235.28	3314.46	-2.39	-	-