University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Parallel Performance Analysis of Large Markov Models

William J. Knottenbelt

Beit Fellow for Scientific Research

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, December 1999

Abstract

Stochastic performance models provide a formal way of capturing and analysing the complex dynamic behaviour of concurrent systems. Such models can be specified by several high-level formalisms, including Stochastic Petri nets, Queueing networks and Stochastic Process Algebras. Traditionally, performance statistics for these models are derived by generating and then solving a Markov chain corresponding to the model's behaviour at the state transition level. However, workstation memory and compute power are often overwhelmed by the sheer number of states in the Markov chain and the size of their internal computer representations.

This thesis presents two parallel and distributed techniques which significantly increase the size of the models that can be analysed using Markov modelling. The techniques attack the space and time requirements of both major phases of the analysis, i.e. construction of the Markov chain from a high-level model (state space generation) and solution of the Markov chain to determine its equilibrium distribution (steady-state solution). Space requirements are reduced through the use of probabilistic and disk-based storage schemes. Time requirements are reduced by exploiting the compute power provided by a distributed memory parallel computer or a network of workstations. Neither method places any restrictions on the type of model that can be analysed.

Both techniques have been implemented in C++ on a 16-node Fujitsu AP3000 distributed memory parallel computer. The methods are applied to the analysis of very large models of the order of 100 million states and 1 billion transitions. The state generator delivers substantial speedups (85% efficiency on 16 nodes) and exhibits good scalability which is confirmed by a theoretical performance model. The steady-state analyser delivers competitive speedups for a sparse linear equation solver (45% efficiency on 16 nodes). The other tools required to build a complete parallel analysis pipeline have also been implemented, thus providing an automatic way of obtaining performance measures for unrestricted high-level system specifications with very large underlying Markov chains.

Acknowledgements

I would like to express my thanks and appreciation to:

- My supervisor, Professor Pete Harrison, for his guidance, supervision and expertise, and for his unceasing faith in my ability to handle ever-increasing model sizes.
- My family, for their love, support and encouragement.
- John Darlington, head of the Imperial College Parallel Computer centre, for the use of the Fujitsu AP3000 parallel computer.
- Keith Sephton, AP3000 systems manager, and Steve Newhouse, AP3000 applications manager, for all their help and invaluable technical advice.
- Matt Nichols of Fujitsu Europe, Tom Oliver of FECIT, Nick Maclaren of Cambridge University and Jim Tuccillo of IBM who all tried in various ways to persuade the AP3000 to overlap communication and computation.
- Pieter Kritzinger of the University of Cape Town, for introducing me to the general area of correctness and performance analysis.
- Mark Mestern, also of the University of Cape Town, for investigating some of the initial concepts as part of his M.Sc. dissertation [Mes98].
- My friends Gulden, Silvana and Zully and my lab partners Madhu and Cati, for all their helpful discussion and advice, and for providing a pleasant working environment.
- The trustees of the Beit Fellowship for Scientific Research, for honouring me with the Fellowship and for providing me with funding.

‘I leaned across to Din and told him I was a buyer at 19 000. There was no need to whisper, I could hardly hear myself shout. He looked at me and queried: “What size, Nick?”
“Any size, Fat Boy!”’

Nick Leeson, General Manager of Baring Futures Singapore
June 1992–February 1995.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

The complexity of modern computer and communication systems is on the rise. Whereas the automated systems of the past were usually controlled by one program running on a single machine with a single flow of control, recent years have seen the ascendancy of technologies such as parallel and distributed computing, multi-threaded programming, multi-agent systems and advanced communication networks. Systems based on these technologies typically consist of several co-operating subsystems that execute concurrently and communicate with one another using sophisticated protocols. Familiar examples of systems which exhibit this structure include multiprocessor computers, flexible manufacturing systems, telecommunications networks, railway signalling systems, distributed databases, stock market dealing systems and air traffic control systems.

It is important to assess the performance of these systems before implementation. At best, a failure to meet design goals results in downtime while the system undergoes a costly redesign and/or retrofit. At worst, in the case of safety critical systems, the results can be catastrophic. Unfortunately, attempts to predict the dynamic behaviour of these systems using intuition or "rules of thumb" are doomed to failure because designers cannot foresee the many millions of possible interactions between components. Consequently the likelihood of problems caused by subtle bugs like race conditions is high.

A more rigorous engineering approach which allows for the mechanical verification of performance properties at design time is based on the construction and analysis of a *model* of the system. A model is a simplified representation of the real world which captures essential aspects of the system's dynamic behaviour. Models are specified using graphical or alphanumerical languages known as *modelling formalisms*.

There are two main approaches to obtaining performance statistics from a model: *simulation* and *analytical* methods. Simulation is used to model systems at arbitrary levels of detail, producing inexact results bounded by confidence intervals. However, there is a high cost and effort involved in constructing accurate models and the length of execution time required to produce reliable results can be very long. Analytical models, on the other hand, make use of formal, abstract models from which exact results can be obtained by generating and solving a set of equations derived from the model.

This thesis focuses on a widely-used analytical performance modelling technique known as *Markov chain modelling*. Markov chains model the low-level stochastic behaviour of a system by describing what possible states the system may enter and how the system moves from one state to another in time. Markov chains are limited to describing systems that have discrete states and which satisfy the fundamental property that the future behaviour of the system depends only on the current state. Despite these limitations, Markov chains are flexible enough to model the phenomena found in complex concurrent systems such as blocking, synchronisation, preemption, state-dependent routing and complex traffic arrival processes. In addition, tedious manual enumeration of all possible system states and transitions in a Markov chain is not necessary. Instead, chains can be automatically derived from several widely-used high-level modelling formalisms such as Generalised Stochastic Petri nets, queueing networks, Queueing Petri nets and Stochastic Process Algebras.

A major difficulty associated with Markov modelling, is the *state explosion problem*, whereby workstation memory and compute power are overwhelmed by the sheer number of the states that emerge from complex models and by the large amount of storage required to represent individual states. Consequently, a major challenge and focus of research is the development of methods and data structures which minimize the memory and runtime required to generate and solve very large Markov chains. A common approach to this "largeness" problem is to restrict the structure of the models that can be analysed. This allows for the application of efficient techniques which exploit the restricted structure. We do not adopt this approach, however, preferring to seek unrestricted scalable parallel and distributed algorithms which leverage the compute power, memory and disk space of several processors.

Our goal is a complete parallel analysis pipeline which allows for the automatic derivation of performance objectives starting from a high-level system specification. In addition to the core state generation and steady-state solution modules, such a pipeline requires a flexible interface language front-end for system description, and a performance analyser back-end that combines the low-level results given by the steady-state distribution and the state space into more meaningful higher-level performance measures such as rewards and throughput measures.

1.2 Contributions

This thesis presents two novel parallel and distributed techniques for the generation and solution of very large Markov chains derived from stochastic models. In contrast to many contemporary techniques for large Markov chains, the algorithms do not require models to conform to a restricted structure or hierarchy, nor do they require the state space to exhibit any regularity.

The first technique is a parallel dynamic probabilistic state space exploration algorithm that generates a system's underlying Markov chain by enumerating the possible states that the system may enter. The algorithm stores compressed hash signatures instead of full state descriptors, so it has a very low memory usage that is independent of the size of individual state vectors. The memory savings come at a cost since there is a non-zero probability that distinct states may be misidentified if they have the same compressed representation. However, this probability can be quantified and can be made arbitrarily small. In addition, the algorithm parallelises well, exhibiting excellent load balance, good speedups and good scalability. The unique 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. Careful analysis of the communication patterns leads to an enhanced version of the algorithm which has a lower communication overhead and which delivers even better speedups. Theoretical performance models of both state generation algorithms confirm their good scalability properties.

Having established the capability to generate large Markov chains, the second technique developed is a parallel disk-based method which solves chains for their steady-state distribution. The focus is on two scalable numerical methods for the solution of large systems of linear equations, namely the Jacobi and Conjugate Gradient Squared (CGS) algorithms. Parallel sparse matrix-vector multiplication emerges as the critical bottleneck in these methods. Exploitation of the data locality typically found in automatically generated transition matrices leads to an efficient disk-based matrix-vector multiply kernel that is characterised by low per-processor memory usage, low communication cost and good load balance. This kernel is embedded into a distributed high-performance solver software architecture which makes use of two processes per node to allow for the overlap of disk I/O with computation and communication. At a slightly higher memory cost, the kernel allows for the complete overlap of disk I/O, communication and computation.

The generation and solution techniques are implemented on a 16-node Fujitsu AP3000 distributed memory parallel computer, along with the remaining tools required for a complete parallel performance analysis pipeline. The resulting pipeline delivers good speedups and provides the capability to automatically derive performance measures for models with underlying Markov chains of 100 million states and 1 billion transitions.

1.3 Thesis Outline

The layout of the rest of this thesis is as follows:

Chapter 2 presents background material. The relationship between physical systems and stochastic models is discussed and four standard paradigms for specifying stochastic models are reviewed. An overview of Markov theory and the principles of iterative techniques for solving systems of linear equations is presented.

Chapter 3 considers several contemporary approaches for the analysis of large Markov chains. The first section reviews current methods for state space generation, including probabilistic, symbolic, reduction-based and parallel approaches. The second section surveys the latest methods for the solution of linear systems derived from Markov models, including Kronecker, symbolic, "on-the-fly", disk-based and parallel techniques.

Chapter 4 gives details of the new parallel state space generation algorithm. A dynamic hash compaction storage scheme is described, analysed in terms of its reliability and space complexity, and compared to existing probabilistic techniques. The storage scheme is then integrated into a parallel state generation algorithm. Analysis of communication patterns leads to an enhanced parallel algorithm that has a lower communication overhead. Theoretical performance models of the original and enhanced state space generation algorithms quantify their benefits in terms of distributed runtime, speedup and efficiency. The good speedups and scalability predicted by the theoretical performance models are confirmed by observed results from an implementation running on a Fujitsu AP3000 distributed memory parallel computer. Finally, the chapter discusses suitable choices for the three hash functions upon which our algorithms are based.

Chapter 5 develops a parallel disk-based technique for determining the steady-state probability distributions of very large Markov chains. The focus is on two scalable numerical methods, namely the Jacobi and Conjugate Gradient Squared algorithms. The efficiency of the parallel sparse matrix-vector multiplication operation that is central to both methods is improved in two ways. Firstly, communication costs are reduced by permuting the rows and columns of the transition matrix to exploit the structure induced by distributed breadth-first search state generators. Secondly, a good load balance is achieved by partitioning the transition matrix over processors such that each processor is assigned the same number of non-zero elements. A distributed matrix-vector multiplication kernel based on this mapping is outlined, and integrated into a high-performance software architecture for a parallel disk-based Markov chain solver. The architecture makes use of two processes per node to overlap disk I/O with communication and computation. Numerical results from an implementation of the solver running on a Fujitsu AP3000 distributed memory parallel computer show good speedups and the capability to analyse extremely large models.

Chapter 6 describes the implementation of a complete parallel performance analysis pipeline, comprising an interface language parser, a parallel state space generator, a parallel matrix transposer, a parallel disk-based steady-state solver and a parallel performance analyser. This pipeline provides the ability to automatically derive performance statistics (such as mean buffer occupancy or data throughput) for a wide variety of high-level models. The implementation is in C++ and uses the Message Passing Interface (MPI) standard. Each major module in the pipeline is discussed, concentrating on data structures, algorithms, and the results of various optimization experiments.

Chapter 7 summarises the main contributions of the thesis, discusses applications, presents conclusions and considers future work.

Appendix A describes the two scalable high-level models that are used as demonstration examples in Chapters 4 and 5.

Appendix B outlines the architecture of the Fujitsu AP3000 distributed memory parallel computer, and presents an accurate interprocessor communication cost model. The appendix also highlights a problem with the machine's ability to overlap communication and computation.

1.4 Statement of Originality

I declare that this thesis was composed by myself, and that the work that it presents is my own except where otherwise stated. Some of the background material has been previously presented as part of my M.Sc. dissertation [Kno96], which concerned the development of a sequential performance analyser for timed transition systems.

1.5 Publications

The material in Chapter 4 was presented at the 10th International Conference on Modelling, Techniques and Tools (TOOLS '98) in Palma de Mallorca, Spain in September 1998 [KMHK98]. An extended version of this paper was one of six papers from the conference invited to appear in a Special Edition of Performance Evaluation [KHMK99].

Chapter 5 formed the basis of a paper which was presented at the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99), in Zaragoza, Spain in September 1999 [KH99].

Chapter 2

Background Theory

2.1 Introduction

This chapter presents background theory on the subject of stochastic modelling. We begin by discussing how the performance of physical systems can be analysed using stochastic models. Four standard paradigms for specifying stochastic models, viz. Stochastic Petri nets, Queueing networks, Queueing Petri nets and Stochastic Process Algebras are reviewed. We refer mainly to [BK95], [Rei92], [Hoc96], [HP93] and [Hil94].

Next we present an overview of Markov theory, which is the vehicle we use to obtain performance statistics. Markov theory has been extensively reviewed in the literature; here we refer to [Kle75], [KS60], [Ste94], [Goo88] and [BDMC+94].

Determining the steady-state distribution of a Markov chain requires the ability to solve a potentially very large system of linear equations. We therefore conclude this chapter by reviewing two classes of iterative solution techniques for solving general systems of linear equations, viz. classical iterative methods and Krylov subspace techniques. We refer to [Var62], [HY81], [Ste94] and [Wei95].

2.2 Physical Systems and Stochastic Models

Physical systems are usually very complex and detailed. In the early design stages, it is therefore usually infeasible to construct a prototype or even a detailed simulation of a system in order to study its performance, especially if several design alternatives are being evaluated. Instead, it is often cheaper and quicker to construct and analyse a high-level *stochastic model* of the system under consideration. A stochastic model is a simplified representation of a physical system that includes only essential aspects of the system's dynamic behaviour. Determining the right level of detail for the model, such that the results are sufficiently accurate while the computational cost of analysing the model is reasonable, is a subtle art that requires experience and ingenuity. Often, however, even simple models are adequate *to provide insight into trends and to assess the relative performance of several design alternatives.*

A stochastic model has one or more attributes which jointly characterise the behaviour of the system it represents. These attributes may take on different values as the system evolves. A vector of these attributes, known as the *state descriptor vector*, characterises the configuration or *state* of the system at any point in time. All possible states of the system may be obtained by a brute force enumeration of all reachable values of the state descriptor starting from some initial state. The resulting *state graph* describes how the system moves from state to state and is isomorphic to a Markov Chain which can be solved (under certain standard conditions) to determine the long-run proportion of time the system spends in each state.

2.3 Modelling Formalisms

Stochastic models are specified using alphanumerical or graphical languages known as *modelling formalisms.* This section describes four popular formalisms: Generalised Stochastic Petri nets, queueing networks, Queueing Petri nets and Stochastic Process Algebras. For each formalism, we describe what constitutes a *state* of the system and define a *state descriptor vector*. We also discuss the advantages and disadvantages of each methodology.

2.3.1 Generalised Stochastic Petri Nets (GSPNs)

Petri nets are a graphical modelling formalism for describing the behaviour of concurrently-executing asynchronous processes. Systems that have been successfully modelled with Petri nets include communication protocols, parallel programs, multiprocessor memory caches and distributed databases [Pet81, Rei92].

The simplest type of Petri nets are Place-Transition nets, which were originally conceived by Carl Adam Petri as a formalism for establishing the correctness of concurrent systems. Place transition nets are directed graphs with two types of node:

- *places*, drawn as circles, which model conditions or objects. Inside the places are *tokens*, drawn as black dots, which represent the specific value of a condition or object. A particular arrangement of the tokens across all the places is known as a *marking*.
- *transitions*, drawn as rectangles, which model activities that change the values of conditions and objects.

Arcs are drawn between places and transitions and vice versa to specify how objects are changed by a certain activity.

A Petri net is defined by its structure and an initial marking. The formal definition of a Petri net is as follows:

Definition 2.1 *A* Place-Transition net *is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$ *where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places.*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions.*

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

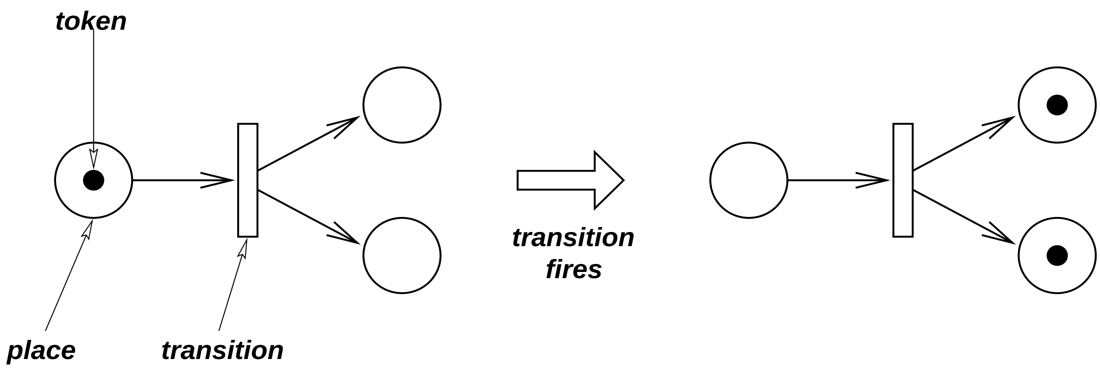


Figure 2.1: Petri net transition firing

The dynamic behaviour of a Place-Transition net is determined by the *enabling* and *firing* of transitions, which results in a flow of tokens between places. A transition is enabled when each of its input places has at least one token on it. An enabled transition may fire, removing a token from each input place and depositing a token on each output place, as shown in Fig. 2.1.

Definition 2.2 *The formal rules for the enabling and firing of transitions are as follows [BK95]:*

1. *A* **marking** *of a Place-Transition net is a function* $M : P \rightarrow \mathbb{N}_0$. $M(p)$ *denotes the number of tokens in place* $p \in P$.
2. *A transition* $t \in T$ *is* **enabled** *at M, iff* $M(p) \geq I^-(p,t), \forall p \in P$.
3. *A transition* $t \in T$*, enabled at marking* M*, may* **fire** *yielding a new marking* M' *where*

$$M'(p) = M(p) - I^-(p,t) + I^+(p,t), \forall p \in P$$

We say M' is **directly reachable** *from M and write $M \rightarrow M'$. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$. A marking M' is* **reachable** *from M iff $M \rightarrow^* M'$.*

Each distinct marking of a Petri net that is reachable from some initial marking M_0 corresponds to a state of the system. Thus the concepts of state and marking are interchangeable in the context of Petri nets and a suitable state descriptor is:

$$M = (p_1, p_2, \ldots, p_{|P|})$$

The set of all markings that are reachable from M_0 constitute the *state space* or *reachability set* of the Petri net. The connections between states form the *state graph*, also known as the *reachability graph*.

From the reachability graph and the structure of the Petri net, Place-Transition nets can be used to test a system for desirable correctness characteristics such as freedom from deadlock, liveness and boundedness. However, since Place-Transition nets do not include a notion of time, it is not possible to model performance. Consequently, several classes of time-augmented Petri nets have been developed, either by attaching time delays to transition firings, or by specifying sojourn times of tokens on places.

One of the most flexible and widely used time-augmented Petri net representations are Generalised Stochastic Petri nets (GSPNs) [ACB84]. GSPNs have two types of transitions: *immediate* transitions and *timed* transitions. Once enabled, immediate transitions fire in zero time, while timed transitions fire after an exponentially distributed firing delay. Firing of immediate transitions has priority over the firing of timed transitions.

The formal definition of a GSPN is as follows:

Definition 2.3 *A GSPN is a 4-tuple $GSPN = (PN, T_1, T_2, W)$ where*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net.*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$,

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

    or
  - *a (possibly marking dependent)* **weight** $\in \mathbb{R}^+$ *specifying the relative firing frequency, when transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$*.*

If several immediate transitions are simultaneously enabled in a given marking, then their weights are used to determine the relative frequency with which each transition fires. In particular, given n simultaneously enabled immediate transitions $t_1, t_2, \ldots, t_n$ with corresponding weights $w_1, w_2, \ldots, w_n$, transition t_i is assumed to fire with probability $w_i / \sum_{k=1}^{n} w_k$.

The state space of a GSPN contains two types of markings. Since immediate transitions fire in zero time, the sojourn time in markings which enable immediate transitions is zero. Such markings are called *vanishing* markings because these states will never be observed by an observer who randomly examines the stochastic process of a GSPN, even though the stochastic process sometimes visits them. On the other hand, markings which enable timed transitions only will have an exponentially distributed sojourn time. Such markings are not left immediately and are referred to as *tangible* markings.

Since no time is spent in vanishing markings, vanishing markings have no effect on the resulting performance statistics derived for a GSPN, so they are often eliminated during state space generation [CMT91, Kno96]. This approach reduces both the size of the state space and state graph and also the number of states that have to be stored during state space exploration. The resulting *tangible reachability graph* (TRG) is isomorphic to a continuous time Markov chain (CTMC), allowing performance measures to be obtained by numerical solution

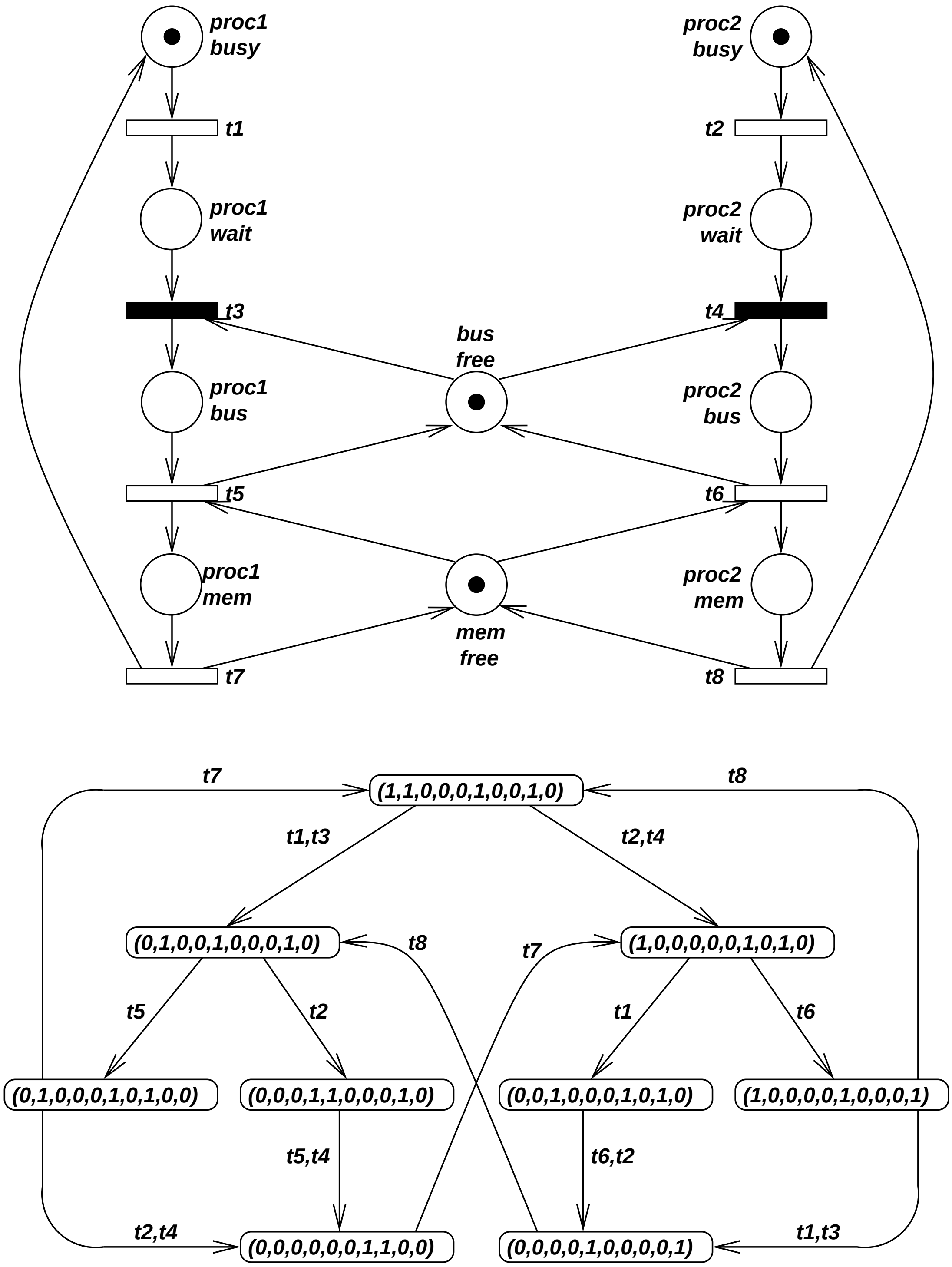


Figure 2.2: A GSPN model and its underlying tangible reachability graph

of the underlying chain. Fig. 2.2 shows a simple GSPN model of a dual processor machine and its underlying tangible reachability graph (with vanishing states eliminated).

The graphical representations of GSPNs becomes complex for realistic models. One way of reducing this complexity is to distinguish between individual tokens. Coloured GSPNs (CGSPNs) [DCB93] make this distinction by attaching *colour* to tokens and by defining *firing modes* on transitions.

Before formally defining a CGSPN, we must first define *multisets* and *Coloured Petri nets* (CPNs). CPNs are the coloured variants of Place-Transition nets on which CGSPNs are based.

Definition 2.4 *A* **multiset** m *over a non-empty set* S*, is a function* $m \in [S \rightarrow \mathbb{N}_0]$. *The non-negative integer* $m(s) \in \mathbb{N}_0$ *is the number of appearances of the element* s *in the multi-set* m.

Definition 2.5 *A* **Coloured Petri net** *(CPN) is a 6-tuple* $CPN=(P,T,C,I^-,I^+,M_0)$*, where*

- P *is a finite and non-empty set of places,*
- T *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$,
- C *is a colour function defined from* $P \cup T$ *into finite and non-empty multisets,*
- I^- *and* I^+ *are the backward and forward incidence functions defined on* $P \times T$ *such that* $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)], \forall\ (p,t) \in P \times T$,
- M_0 *is a function defined on* P *describing the initial marking such that* $M_0(p) \in C(p), \forall p \in P$.

A marking M of a CPN is described in terms of the number of tokens of each colour on each place. In particular, the multiset of tokens on place $p \in P$ is denoted by $M(p)$. By the definition of a multiset, $M(p)(c) \rightarrow \mathbb{N}_0$ denotes the number of tokens on place $p \in P$ of colour $c \in C(p)$.

Definition 2.6 *The dynamic behaviour of a CPN is given as follows:*

1. *A transition $t \in T$ is* **enabled** *in a marking M w.r.t. a colour $c' \in C(t)$, denoted by $M[(t, c') >$, iff $M(p)(c) \geq I^-(p,t)(c')(c), \forall p \in P, c \in C(p)$.*

2. *An enabled transition $t \in T$ may* **fire** *in a marking M w.r.t. a colour $c' \in C(t)$ yielding a new marking M', denoted by $M \rightarrow M'$ or $M[(t, c') > M'$, with*
   $M'(p)(c) = M(p)(c) + I^+(p,t)(c')(c) - I^-(p,t)(c')(c), \forall p \in P, c \in C(p)$.

Definition 2.7 *A* **Coloured GSPN** *(CGSPN) is a 4-tuple* $CGSPN = (CPN, T_1, T_2, W)$ *where*

- $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Coloured Petri net.*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$,
- $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$,
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry w_i is a function of* $[C(t_i) \rightarrow \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ *is either*
  - *a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay with respect to colour c, if $t_i \in T_1$*
    or
  - *a (possibly marking dependent) firing weight with respect to colour c, if $t_i \in T_2$.*

While CGSPNs provide a more compact representation than GSPNs, they do not in fact have any additional modelling power over GSPNs, since every CGSPN may be uniquely unfolded into a GSPN representing the same model.

GSPNs provide a natural way of modelling features such as simultaneous resource possession and synchronisation, but several difficulties arise when attempting to model queues. Even simple scheduling strategies like FCFS are difficult to represent with low-level Petri net elements. In addition, advance knowledge of the maximum number of customers in a queue is required and it is difficult to model service times given by complex distributions such as a Coxian distribution.

2.3.2 Queueing Networks

Queueing networks are a widely-used performance analysis technique for systems which can be naturally represented as a collection of interconnected queues. Systems which have been successfully analysed with queueing networks include computer systems and communications networks.

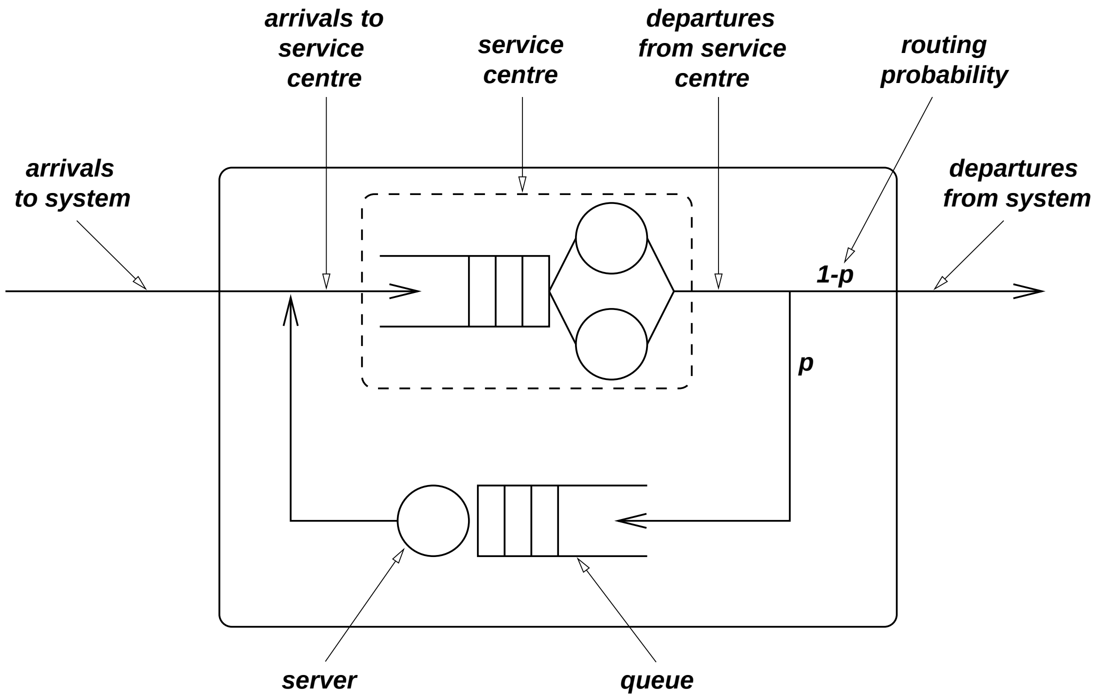


Figure 2.3: A simple queueing network

As shown in Fig. 2.3, a queueing network consists of three types of components:

- *Service centres*, each of which consists of one or more *queues* and one or more *servers*. The servers represent the resources of the system available to serve customers. An arriving customer will be served immediately if a free server can be allocated to the customer or if a customer in service is preempted. Otherwise, the customer must wait in one of the queues until a server becomes available.

- *Customers*, who demand service from the service centres and which represent the load on the system. Usually customers are grouped into classes, where customers in one class exhibit similar behaviour and place similar demands on service centres.

- *Routes*, which are the paths workloads follow through a network of service centres. The routing of customers is determined by *routing probabilities* which may be dependent on the state of the network and customer class. If the routing is such that no customers may enter or leave the system, the system is said to be *closed*. If customers arrive externally and eventually depart, the system is said to be *open*. If some classes of customers are closed and some are open, then the system is said to be *mixed*.

To be fully specified, a queueing network requires the following parameters to be defined:

- The *number of service centres*.

- The *number of queues* at each service centre. For each of these queues we further need to define:

  - The *capacity of each queue*, which may be infinite or finite with capacity k.

  - The *queue scheduling discipline*, which determines the order of customer service. Different customer classes may have different scheduling priorities. Common scheduling rules include First-Come First-Served (FCFS), Last-Come First-Served Preemptive-Resume (LCFS-PR), Round Robin (RR) and Processor Sharing (PS).

  - For open classes of customers, we need to define an *input source process* specifying the interarrival time distribution of each customer class at each queue. For a Poisson arrival process with rate λ, this distribution is given by an exponential distribution with parameter λ.

- The *number of servers* at each service centre. For each of these servers we further we need to define the *service time distribution* for each customer class at each server. This is often exponential. More general distributions can be approximated using a Coxian or other phase-type distribution [Ste94, pg. 51–52].

- The *routing probability matrix* for each customer class. This matrix specifies the probabilistic routing of customers between service centres, with the ijth element giving the probability that a customer leaving service centre i will proceed to service centre j. These transitions are assumed to be instantaneous.

The state of individual service centres in a queueing network can be described by a vector. For example, the state of a single-server centre with a Coxian service distribution and blocking may be described by the number of customers in the queue, the phase of service and a parameter to indicate whether or not the server is blocked. The state descriptor of a queueing network as a whole may then be built up by concatenating the vectors describing the state of the individual service centres.

A certain class of queueing networks which satisfy *quasi-reversibility* [Kel79] can be efficiently analysed using so-called product-form solution techniques, the two most well-known of which are Mean Value Analysis (MVA) and the convolution method. Unfortunately, these elegant algorithms fail if one of the prerequisites for the product-form property is violated by the network. In particular, if phenomena such as synchronization, simultaneous resource possession or blocking occur, then usually no product-form queueing network model can be found. In this case, strictly numerical procedures have to be used to obtain exact solutions.

In particular, for finite queueing networks, a tangible reachability graph isomorphic to a Markov chain can be derived and solved for its steady-state distribution (in the same way as for GSPN models).

Queueing networks are often easy to define, parameterise and evaluate. However, their applicability is limited because they lack facilities to describe synchronisation mechanisms.

2.3.3 Queueing Petri Nets (QPNs)

Queueing Petri nets (QPNs) [Bau93] incorporate the concept of queues into a coloured GSPN formalism. In this way, synchronisation mechanisms and queues with various scheduling strategies can be integrated into one model. A QPN extends the concept of a CGSPN by partitioning the set of places into two subsets: *queued places* and *ordinary places*.

A queued place (see Fig. 2.4) consists of two parts: a queue and a depository for tokens which have completed their service at this queue. Tokens, when fired onto a queued place by any of its input transitions, are inserted into the queue according to the scheduling strategy of the queue. Tokens in a queue are not available to transitions. After completion of its service, the token is placed onto the depository. Tokens on this depository are available to all output transitions of the queued place. An enabled timed transition will fire after an exponentially distributed time delay and an immediate transition fires with no delay, as in GSPNs.

The formal definition of a QPN is as follows:

Definition 2.8 *A* **Queueing Petri net** *(QPN) is a triple* $QPN = (CGSPN, P_1, P_2)$ *where:*

- *CGSPN is the underlying Coloured GSPN*
- $P_1 \subseteq P$ *is the set of queued places and*

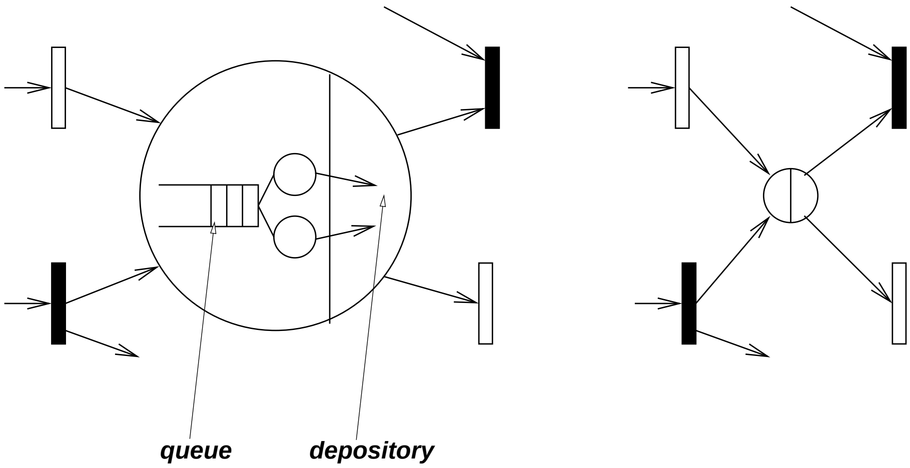


Figure 2.4: A QPN queued place (left) and its shorthand notation (right)

- $P_2 \subseteq P$ *is the set of ordinary places,* $P_1 \cap P_2 = \emptyset$*,* $P = P_1 \cup P_2$*.*

A state or marking M of a QPN consists of two parts $M = (n, m)$ where n specifies the state of all queues and m is the marking of the underlying CGSPN. For a queued place $p \in P_1$, $m(p)$ denotes the marking of the depository. The initial marking M_0 of a QPN is given by $M_0 = (\underline{0}, m_0)$ where $\underline{0}$ is the state denoting that all queues are empty and m_0 is the initial marking of the CGSPN.

There are three possible types of state transitions that may take place in a QPN:

- An enabled immediate transition may fire.
- An enabled timed transition may fire if no immediate transitions are enabled.
- A service in a queued place may complete if no immediate transitions are enabled.

Similar to GSPNs, the firing of immediate transitions has priority over both the firing of timed transitions and the service of tokens in queues. Thus, the state space of a QPN comprises both vanishing and tangible states. By eliminating the vanishing states during or immediately after state space generation, a tangible reachability graph can be constructed in a similar manner as for GSPN models (except that now we use the extended QPN state descriptor and the new QPN firing rules). In the usual way, this graph maps directly onto a Markov chain which may be solved for its steady-state distribution.

QPNs allow for a convenient description of queues within a Petri net paradigm. However, the complexity of the performance analysis, as determined by the size of the state space, is still the same as that obtained by modelling the queue with GSPN elements.

2.3.4 Stochastic Process Algebras (SPAs)

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

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

Two of the best known process algebras are Hoare's Communicating Sequential Processes (CSP) [Hoa85] and Milner's Calculus of Communicating Systems (CCS) [Mil89]. These algebras do not include a notion of time so they can only be used to determine qualitative correctness properties of systems such as the freedom from deadlock and livelock. Stochastic Process Algebras (SPAs) additionally allow for quantitative performance analysis by associating a random variable, representing duration, with each action.

Here we will briefly describe the Markovian SPA PEPA [Hil94]. Other SPAs include TIPP [RS94, HR94] and MPA [Buc94b] which are similar to PEPA, while non-Markovian SPAs include SPADES [Str93, HS99] and EMPA [BDG94]. PEPA models are built from components which perform activities of form (α, r)

where α is the action type and $r \in \mathbb{R}^+ \cup \top$ is the exponentially distributed rate of the action. The special symbol $\top$ denotes an passive activity that may only take place in synchrony with another action whose rate is specified.

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

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

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

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

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

Cooperation over the empty set $P \underset{\emptyset}{\bowtie} Q$ represents the independent concurrent execution of processes P and Q and is denoted as $P||Q$.

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

PEPA specifications can be mapped onto continuous time Markov chains in a straightforward manner. Based on the labelled transition system semantics that

$$
\begin{array}{lcllcllcl}
Proc & \stackrel{\mathrm{def}}{=} & (think, p_1\lambda).Proc_1 + (think, p_2\lambda).Proc_2 & & & & & & \\
Proc_1 & \stackrel{\mathrm{def}}{=} & (local, m).Proc & & & & & & \\
Proc_2 & \stackrel{\mathrm{def}}{=} & (get, g).Proc_3 & Proc_3 & \stackrel{\mathrm{def}}{=} & (use, \mu).Proc_4 & Proc_4 & \stackrel{\mathrm{def}}{=} & (rel, r).Proc \\
Mem & \stackrel{\mathrm{def}}{=} & (get, \top).Mem_1 & Mem_1 & \stackrel{\mathrm{def}}{=} & (use, \mu).Mem_2 & Mem_2 & \stackrel{\mathrm{def}}{=} & (rel, \top).Mem \\
Sys_4 & \stackrel{\mathrm{def}}{=} & Proc \underset{L}{\bowtie} Mem & L = \{get, rel, use\} & & & & &
\end{array}
$$

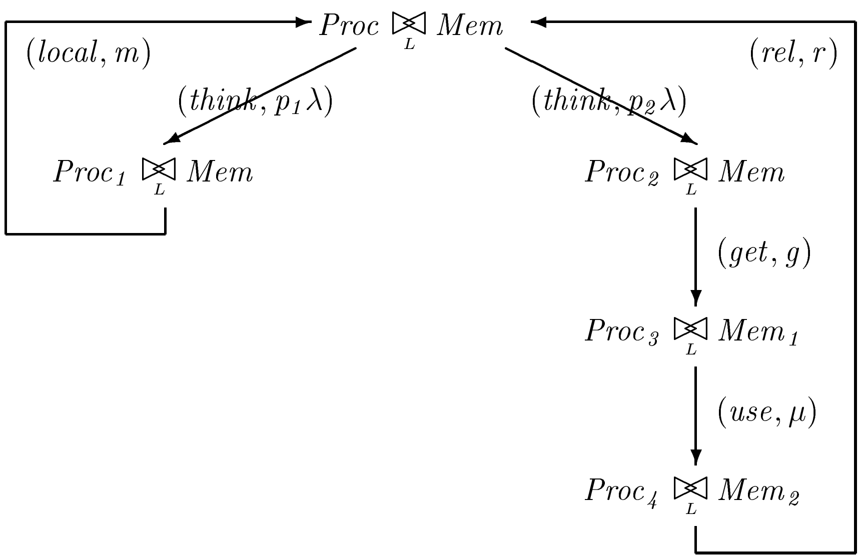


Figure 2.5: A PEPA specification and its corresponding derivation graph [HR98]

are normally specified for a process algebra system, a transition diagram or *derivation graph* can be associated with any language expression. This graph describes all possible evolutions of a component and, like a reachability graph in the context of GSPNs, is isomorphic to a CTMC which can be solved for its steady-state distribution. Fig. 2.5 shows a PEPA specification of a multiprocessor system together with its corresponding derivation graph.

The main advantage of process algebras over other formalisms is their support for *compositionality*, i.e. the ability to construct complex models in a stepwise fashion from smaller building blocks, and *abstraction*, which provides a way to treat components as black boxes, making their internal structure invisible. However, unlike the other formalisms we have considered, process algebras lack an intuitive graphical notation so do not always present a clear image of the dynamic behaviour of the model [Hil94].

2.4 Markov Theory

2.4.1 Stochastic Processes

As we have mentioned, the behaviour of a system can often be characterised by enumerating all the states that the system may enter and by describing how the system evolves from one state to another over time. In its most general form, such a system can be represented by a *stochastic process.*

Definition 2.9 *A* **random variable** χ *is a variable whose value depends on the outcome of a random experiment. If the value space of* χ *is countable but not necessarily finite, then the random variable is* **discrete** *and its behaviour is characterised by a probability mass function:*

$$p_\chi(x) = P\{\chi = x\}$$

If the value space of χ *is uncountable, then the random variable is* **continuous** *and its behaviour is characterised by a cumulative distribution function:*

$$F_\chi(x) = P\{\chi \leq x\}$$

Definition 2.10 *A* **stochastic process** *is a family of random variables* $\{\chi(t)\}$ *indexed by the time parameter* t*. If the time index set* $\{t\}$ *is countable, the process is a* **discrete-time** *process, otherwise the process is a* **continuous-time** *process. The possible values or states that members of* $\{\chi(t)\}$ *can take on constitute the* **state space** *of the process. If the state space is discrete the process is called a* **chain**.

2.4.2 Markov Processes

In many cases, the future evolution of a system depends only on the current state of the system and not on past history. Stochastic processes for which this property holds are known as *Markov processes.* The *memoryless* behaviour of Markov processes is formally characterised by the *Markov property.*

Definition 2.11 *A stochastic process is a* **Markov process** *satisfying the* **Markov property** *if, for all integers* n *and for any sequence* $t_0, t_1, \ldots, t_n$ *such that* $t_0 \leq t_1 \leq \ldots \leq t_n \leq t_{n+1}$ *we have*

$$P\{\chi(t_{n+1}) \leq x \mid \chi(t_n) = x_n, \chi(t_{n-1}), \ldots, \chi(t_0)\} = P\{\chi(t) \leq x \mid \chi(t_n) = x_n\}$$

The Markov property requires that the next state can be determined knowing nothing other than the current state, not even how much time has been spent in the current state. A consequence of the Markov property is that the sojourn time τ spent in a state must satisfy:

$$P\{\tau \geq s + t \mid \tau \geq t\} = P\{\tau \geq s\} \quad \forall\, s, t \geq 0 \tag{2.1}$$

As we shall see, this condition places restrictions on the distribution of time spent in a state.

Definition 2.12 *A* **homogeneous Markov chain** *is a Markov chain whose probabilities are stationary with respect to time. That is:*

$$P\{\chi(t) \leq x \mid \chi(t_n) = x_n\} = P\{\chi(t - t_n) \leq x \mid \chi(0) = x_n\}$$

Discrete-time Markov Chains

A discrete-time Markov chain (DTMC) is a Markov process with a discrete state space which is observed at a discrete set of times. Without loss of generality, we can take the time index set $\{t\}$ to be the set of counting numbers $\{0, 1, 2, \ldots\}$. The observations at these times define the random variables $\chi_0, \chi_1, \chi_2, \ldots$ at time steps $0, 1, 2, \ldots$ respectively.

Definition 2.13 *The variables $\chi_0, \chi_1, \ldots$ form a* **discrete-time Markov chain** *if for all n $(n = 0, 1, 2, \ldots)$ and all states x_n we have:*

$$\begin{aligned} P\{\chi_{n+1} = x_{n+1} \mid \chi_0 = x_0, \chi_1 = x_1, \ldots, \chi_n = x_n\} \\ = P\{\chi_{n+1} = x_{n+1} \mid \chi_n = x_n\} \end{aligned}$$

A homogenous discrete-time Markov chain may be represented by a one-step transition probability matrix P with elements:

$$p_{ij} = P\{\chi_{n+1} = x_j \mid \chi_n = x_i\}$$

where χ_t represents the state of the system at discrete time-step $t \in \mathbb{N}$. That is, p_{ij} gives the probability of x_j being the next state given that x_i is the current state. Note that the entries of P must satisfy:

$$0 \leq p_{ij} \leq 1 \ \forall\ i, j \quad \text{and} \quad \sum_j p_{ij} = 1 \ \forall\ i$$

Definition 2.14 *Let S_0 denote a subset of the state space S, and $\overline{S_0}$ its complement. Then S_0 is* **closed** *or* **final** *if no single-step transition is possible from any state in S_0 to any state in $\overline{S_0}$.*

Definition 2.15 *A Markov chain is* **irreducible** *if every state can be reached from every other state. Otherwise, the state space contains one or more* **closed** *subsets of states and the chain is* **reducible**.

Let $f_j^{(m)}$ denote the probability of leaving state x_j and first returning to that same state in m steps. Then the probability of ever returning to the state x_j is given by:

$$f_j = \sum_{m=1}^{\infty} f_j^{(m)}$$

Definition 2.16 *For any state x_j:*

- *if $f_j = 1$ then x_j is* **recurrent**; *else*
- *x_j is* **transient**.

Definition 2.17 *State x_j is* **periodic** *with period η if the Markov chain can return to state x_j only at a time step in the set $\{\eta, 2\eta, 3\eta, \ldots\}$ where $\eta \geq 2$ is the smallest such integer. If $\eta = 1$ then x_j is* **aperiodic**.

Definition 2.18 *The* **mean recurrence time** *of recurrent state x_j is*

$$M_j = \sum_{m=1}^{\infty} m f_j^{(m)}$$

which is the average number of steps taken to return to state x_j for the first time after leaving it. If $M_j = \infty$, state x_j is **recurrent null**; *otherwise x_j is* **recurrent nonnull** *(also known as* **positive recurrent***).*

Theorem 2.1 *The states of an irreducible Markov chain are either all transient or all recurrent nonnull or all recurrent null. If the states are periodic, then they all have the same period [Kle75, pg. 29].*

The most important part of Markov chain analysis is to determine how much time is spent in each of the states x_j. For a discrete-time Markov chain, we define:

$$\pi_j^{(m)} = P\{\chi_m = x_j\}$$

as the probability of finding the Markov chain in state x_j at time step m.

Definition 2.19 *Let z be a vector describing a probability distribution whose elements z_j denote the probability of being in state x_j. Then, z is a* **stationary probability distribution** *of a DTMC with one-step transition matrix P if and only if $zP = z$.*

Definition 2.20 *The* **limiting probability distribution** *$\{\pi_j\}$ of a discrete-time Markov chain is given by:*

$$\pi_j = \lim_{m \to \infty} \pi_j^{(m)}$$

Note that the existence of a stationary distribution of a Markov chain does not necessarily imply the existence of a limiting probability distribution, and vice versa. The next theorem addresses the issue of when the limiting and stationary probabilities exist.

Theorem 2.2 *In an irreducible and aperiodic homogeneous Markov chain, the limiting probabilities $\{\pi_j\}$ exist and are independent of the initial probability distribution. Moreover one of the following conditions hold:*

- *Every state x_j is transient or every state x_j is recurrent null, in which case $\pi_j = 0$ for all x_j and there exists no stationary distribution (even though the limiting probability distribution exists). In this case, the state space must be infinite.*

- *Every state x_j is recurrent nonnull with $\pi_j > 0$ for all x_j, in which case the set $\{\pi_j\}$ is a limiting and stationary probability distribution and*

$$\pi_j = \frac{1}{M_j}$$

In this case the π_j are uniquely determined from the set of equations:

$$\sum_i \pi_j = \pi_i p_{ij} \quad \textit{subject to} \quad \sum_i \pi_i = 1 \tag{2.2}$$

If $\pi = (\pi_1, \pi_2, \ldots)$ is a vector of limiting probabilities, Eq. 2.2 can be rewritten as

$$\pi = \pi P$$

where P is the transition probability matrix. The vector π is called the *steady-state solution* of the Markov chain.

If the state space of the Markov chain is finite (which we will always assume is the case), the chain is called *finite*; if, in addition, the chain is irreducible and recurrent nonnull, then it is *ergodic*.

Continuous-time Markov Chains

A continuous-time Markov chain (CTMC) is a Markov process with a discrete state space and a state that may change at any real-valued time instant. Its defining characteristic is that it possesses the Markov property of Def. 2.11 at all time indices.

For the case of a continuous-time Markov chain, the only sojourn time distribution which satisfies the sojourn time condition of Eq. 2.1 is the exponential distribution.

A homogeneous continuous-time Markov chain may be represented by a set of states and an *infinitesimal generator matrix* Q where q_{ij}, represents the transition rate between states x_i and x_j for $i \neq j$. The parameter of the exponential distribution of the sojourn time in state x_i is given by $-q_{ii}$ where $q_{ii} = -\sum_{j \neq i} q_{ij}$. Note that, by construction, the entries of Q must satisfy:

$$\sum_j q_{ij} = 0 \quad \forall\ i$$

Definition 2.21 *Let z be a vector describing a probability distribution whose elements z_j denote the probability of being in state x_j. Then, z is a* **stationary probability distribution** *of a CTMC with infinitesimal generator matrix Q if and only if $zQ = 0$.*

Definition 2.22 *The* **limiting probability distribution** $\{\pi_j\}$ *of a continuous-time Markov chain is given by:*

$$\pi_j = \lim_{t\to\infty} P\{\chi(t) = x_j\}$$

Theorem 2.3 *In a finite, homogeneous, irreducible, continuous-time Markov chain, the limiting probabilities* $\{\pi_j\}$ *always exist and are independent of the initial probability distribution. Moreover, the set* $\{\pi_j\}$ *is also a stationary probability distribution which can be uniquely determined by solving the set of equations:*

$$q_{jj}\pi_j + \sum_{i\neq j} q_{ij}\pi_i \quad \textit{and} \quad \sum_i \pi_i = 1 \tag{2.3}$$

The set of equations given by Eq. 2.3 is sometimes also referred to as the set of *global balance equations*. In vector form, they may be written as:

$$\pi Q = 0$$

where $\pi = (\pi_1, \pi_2, \ldots)$ is the stationary probability distribution (c.f. Def. 2.21). Equivalently, this system can be written as

$$Q^T \pi^T = 0$$

which allows for the application of general numerical methods for the solution of linear systems of form $Ax = b$.

2.5 Numerical Methods

2.5.1 Classical Iterative Techniques

This section considers some of the oldest and best-known iterative methods for solving linear systems of the form $Ax = b$. These methods are based around matrix splittings of form $A = M - N$ where M is non-singular (i.e. $\det(M) \neq 0$). This splitting is used to define simple iterative schemes of form:

$$x^{(k+1)} = M^{-1}N\,x^{(k)} + c$$

where $c = M^{-1}b$ and neither the iteration matrix $M^{-1}N$ nor c depends on k.

In the case of solving $Ax = 0$ (corresponding to the set of steady state equations $Q^T\pi^T = 0$ derived from a CTMC), the schemes reduce to the form:

$$x^{(k+1)} = M^{-1}N\,x^{(k)}$$

From this equation, the desired solution can be seen to be the eigenvector of the iteration matrix $M^{-1}N$ corresponding to the eigenvalue 1. The convergence of these methods depends on the eigenvalues of the iteration matrix $M^{-1}N$. In particular, the rate of convergence is inversely proportional to the ratio $|\lambda_2|/|\lambda_1|$ where λ_1 and λ_2 are the dominant and the subdominant eigenvalues of the iteration matrix respectively [Bar89]. Consequently, these methods are only guaranteed to converge if the iteration matrix is *primitive*, i.e. if it has one and only one eigenvalue λ_i with $|\lambda_i| = 1$.

The three most commonly used classical iterative methods of this form are presented below.

Jacobi's Method

Jacobi's method [Var62, §3.1] [Ste94, §3.2.2] [HY81, §2.3] is a simple iterative method based on the observation that solving $Ax = b$ is equivalent to finding the solution to the n equations:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \quad i = 1, 2, \ldots, n$$

Now, solving the ith equation for x_i yields:

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}x_j)$$

which suggests the iterative method:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}) \qquad (2.4)$$

where $k \geq 0$ and $x^{(0)}$ is an initial guess at the solution vector.

If we write $A = D - L - U$ where $D = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$ and L and U are strictly lower and upper triangular matrices respectively, Eq. (2.4) can be written in matrix form as:

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b$$

where $D^{-1}(L + U)$ is the iteration matrix characterising the convergence behaviour of the algorithm. The Jacobi method does not always converge for all types of matrices, but one class of matrices for which it is guaranteed to converge is the class of diagonally dominant matrices [KGGK94]. Matrix A is diagonally dominant if and only if $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.

Note that the calculations of the $x_i^{(k)}$s are independent of one another which means equation updates can be performed in parallel.

Gauss-Seidel

The Jacobi method can be improved on by using the newly computed results for x_i as soon as they are available within an iteration. The resulting method is known as the Gauss-Seidel method [Var62, §3.1], [Ste94, §3.2.3] which is given by:

$$x_i^{(k+1)} = (b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)})/a_{ii} \tag{2.5}$$

In matrix form, Eq. 2.5 can be written as:

$$x^{(k+1)} = (D - L)^{-1}(Ux^{(k)} + b)$$

where $(D - L)^{-1}U$ is the iteration matrix characterising the convergence behaviour of the algorithm. This iteration matrix improves considerably on the convergence properties of the Jacobi method. However, as was the case for the Jacobi method, the Gauss-Seidel method is not always guaranteed to converge [KGGK94].

Note that the computations of Eq. 2.5 appear to be serial in nature since the calculations of the $x_i^{(k)}$s now depend on one another. However, if A is sparse and several coefficients are zero, then elements of the new iterate are not necessarily dependent on previous elements. By reordering the equations in this situation, it is sometimes possible to make updates to groups of components in parallel [BBC+94, §3, §4.4].

Successive Overrelaxation (SOR)

Successive Overrelaxation (SOR) [Var62, §3.1] [Ste94, §3.2.4] is an extrapolation technique for accelerating the convergence of the Gauss-Seidel algorithm. The extrapolation works by taking a weighted average of each element of the previous iterate and each element of the newly-computed Gauss-Seidel iterate, i.e.

$$x_i^{(k+1)} = \omega \overline{x}_i^{(k+1)} + (1-\omega) x_i^{(k)} \tag{2.6}$$

where $\overline{x}_i^{(k+1)}$ is the ith element of the newly-computed Gauss-Seidel iterate and $x_i^{(k)}$ is ith element of the previous Gauss-Seidel iterate.

In matrix form, Eq. 2.6 can be written as:

$$x^{(k+1)} = (D - \omega L)^{-1}(\omega U + (1-\omega)D)x^{(k)} + \omega(D - \omega L)^{-1} b$$

with iteration matrix

$$L_\omega = (D - \omega L)^{-1}(\omega U + (1-\omega)D).$$

Note that if $\omega = 1$, the method reduces to the Gauss-Seidel algorithm. In the case $\omega > 1$, we speak of overrelaxation and in the case $\omega < 1$, we speak of underrelaxation. The SOR method converges only for values of ω in the range $0 \leq \omega \leq 2$.

In the case of solving $Ax = 0$, the optimal value of ω is that value which maximizes the difference between the dominant and subdominant eigenvalues of L_ω, thus resulting in the fastest covergence rate. Unfortunately, methods for choosing this optimal value of ω are only known for very restricted classes of matrices [HY81]. Consequently, implementations usually use heuristic adaptive parameter estimation schemes to guess a value for ω which is adjusted every few iterations according to the rate at which the method is converging.

2.5.2 Krylov Subspace Techniques

Krylov subspace techniques [Wei95] [FGN92] [Ste94, §4.3], also referred to as projection methods, are a popular class of iterative methods for solving large systems of linear equations. Many conjugate-gradient type algorithms and their variants are included in this category. They derive their name from the fact that they generate their iterates using a shifted Krylov subspace associated with the coefficient matrix of the system. Before defining a Krylov subspace formally (c.f. Eq. 2.8), we will first provide an overview of the advantages of Krylov subspace techniques.

Krylov subspace techiques have been used to solve systems of linear equations arising from a wide range of scientific and engineering applications including fluid dynamics, atmospheric modelling, structural analysis and finite element analysis. There are three main reasons for this widespread use. Firstly, the methods exhibit good convergence behaviour while being parameter-free. The original conjugate gradient algorithm, for example, provides the same order of convergence as optimal SOR, but without the need for dynamic parameter estimation. Secondly, the methods have become increasingly competitive with classical iterative methods in terms of memory utilization. The most recently developed methods (e.g. CGS [Son89], BiCGSTAB [Vor92], TFQMR [Fre93]) do not require storage of large sequences of vectors (as does GMRES [SS86]), nor do they require multiplication with the transpose of the coefficient matrix (as do BiCG [Fle76], QMR [FN91] and CGNR/CGNE [Yan94]). Finally, the methods are well suited to implementation on parallel computers. Since most Krylov subspace methods compute one or two matrix-vector products and several vector inner products every iteration, the methods can be parallelised by distributing the matrix across processing nodes and using the inner products as synchronization points. For a discussion of the issues involved, see [Saa89] and [GKS95]. In practice, superlinear speedups have been achieved [Bou95], probably as a result of efficient cache utilization.

The development of Krylov subspace techniques began in the early 1950s with the conjugate gradient (CG) algorithm of Hestenes and Stiefel [HS52]. This algorithm is used to solve $n \times n$ linear systems of form $Ax = b$ where A is a symmetric positive definite (SPD) coefficient matrix. Matrix A is said to be positive definite if $x^T Ax > 0$ for all non-zero n-vectors x. The CG method is regarded as an attractive algorithm for two main reasons. Firstly, the algorithm has very modest memory requirements because it uses simple three-term recurrences. Secondly, the algorithm has good convergence properties since the residual is minimized with respect to some norm at each step. The generated residuals are also mutually orthogonal, which guarantees finite termination.

Several algorithms have since been devised to generalise the CG algorithm to allow for arbitrary (i.e. not necessarily symmetric or positive definite) coefficient matrices. Unfortunately, algorithms for non-symmetric coefficient matrices cannot maintain both the short recurrence formulation and the minimization property (see Faber and Manteuffel's paper [FM84] for proof). Thus, by trading off certain optimality conditions against the amount of memory required, three main classes of CG variants have been developed:

- Algorithms which attempt to preserve both properties by transforming a linear system based on a non-symmetric coefficient matrix A into an equivalent system based on the symmetric positive definite matrix $A^T A$ (CGNR) or AA^T (CGNE). This approach is known as conjugate gradient applied to the normal equations.
- "Pure" algorithms for non-symmetric A which are based on maintaining either the short recurrence formulation (e.g. BiCG [Fle76]) or the minimization property (e.g. GMRES [SS86]) but not both.
- "Hybrid" methods for non-symmetric A which seek to combine elements of the short recurrence formulation with minimization properties that are either heuristic (e.g. CGS), localized (e.g. BiCGSTAB) or quasi-optimal (e.g. QMR). This class includes most of the more recently developed CG-type methods such as CGS [Son89], QMR [FN91], BiCGSTAB [Vor92], BiCGSTAB(l) [SF93], and TFQMR [Fre93]).

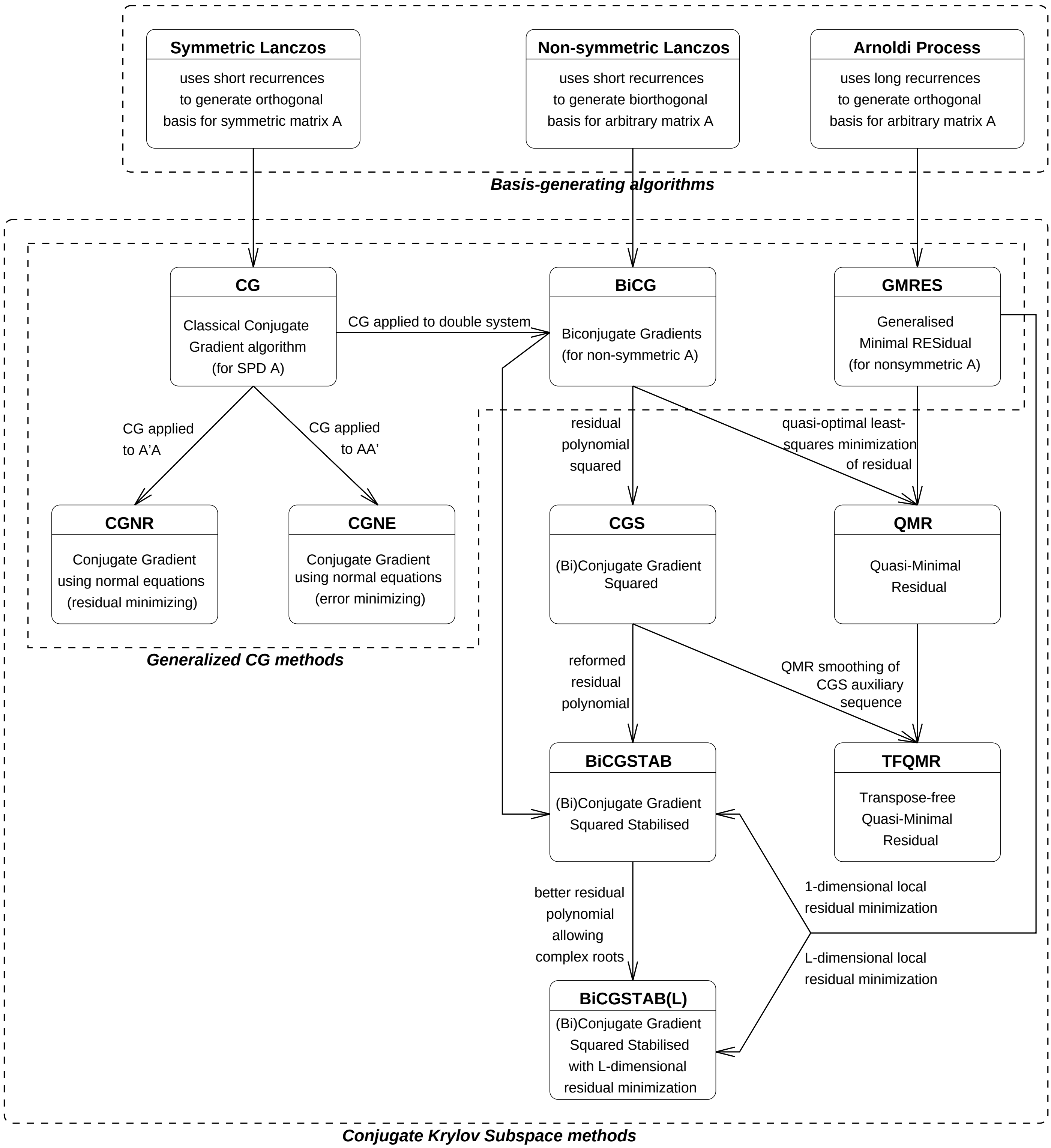


Figure 2.6: An overview of Krylov subspace techniques

Many authors have attempted to resolve the confusion resulting from the development of all these methods by using unifying mathematical frameworks to explore the relationships between them (see e.g. [Wei95], [Wei94], [Gut93a] and [AMS90]). Fig. 2.6 presents a conceptual overview of the most important techniques. The arrows show the relationships between the methods, i.e. how the methods have been generalised from their underlying basis-generating algorithms and also how key concepts have been inherited from one algorithm to the next.

Principles of Krylov Subspace methods

To formalise the concept of a Krylov subspace method, we consider a linear system $Ax = b$ where A is a real $n \times n$ non-symmetric matrix and $x, b \in \mathbb{R}^n$. Let x_0 be an initial guess at the solution vector x with corresponding residual $r_0 = b - Ax_0$. Then, using the notation of Weiss [Wei95], Krylov subspace techniques generate subsequent iterates x_k according to the formula:

$$x_k = x_{k-\sigma_k} + d_k, \quad d_k \in \mathrm{span}(q_{k-\sigma_k,k}, \ldots, q_{k-1,k}) \quad \text{for } k = 1, 2, \ldots \tag{2.7}$$

where $q_{k-i,k} \in \mathbb{R}^n$, σ_k denotes the number of previous q vectors used in the calculation of new iterates and $\mathrm{span}(q_{k-\sigma_k,k}, \ldots, q_{k-1,k})$ is the vector space spanned by all linear combinations of the n-vectors $q_{k-\sigma_k,k}, \ldots, q_{k-1,k}$. Usually all previous q vectors are used i.e. $\sigma_k = k$, in which case we speak of an *exact* method. However, sometimes methods are restarted every σ_{res} steps to limit their memory consumption, i.e. $\sigma_k = (k-1) \bmod \sigma_{res} + 1$. In this case, we speak of a *restarted* method.

The q vectors are generated to fulfill two conditions:

- Firstly, each $q_{k-i,k}$ is a member of the *Krylov subspace*:

  $$\mathcal{K}_{k-i+1}(B, z) \quad = \quad \mathrm{span}(z, Bz, B^2z, \ldots, B^{k-i}z) \tag{2.8}$$

  where B is an $n \times n$ matrix and $z \in \mathbb{R}^n$. For almost all Krylov subspace methods of practical interest (and for all the methods discussed here), $B =$

A and $z = r_0$, i.e. each $q_{k-i,k}$ lies in the Krylov subspace

$$\mathcal{K}_{k-i+1}(A, r_0) \quad = \quad \text{span}(r_0, Ar_0, A^2r_0, ..., A^{k-i}r_0) \tag{2.9}$$

Eq. 2.9 characterises a broad class of methods known as Conjugate Krylov Subspace (CKS) techniques. From the definition of $q_{k-i,k}$ and Eq. 2.7, it follows that

$$x_k \quad \in \quad x_{k-\sigma_k} + \mathcal{K}_k(A, r_0) \tag{2.10}$$

Equivalently,

$$x_k \in \{x_{k-\sigma_k} + d \mid d \in \text{span}(r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0)\}$$

i.e. the iterates lie in a shifted Krylov subspace associated with the coefficient matrix of the system.

- Secondly, the q vectors satisfy the orthogonality condition:

$$r_k^T Z_k q_{k-i,k} = 0 \quad \text{for } i = 1, \ldots, \sigma_k \tag{2.11}$$

where the Z_k are auxiliary non-singular matrices. This equation can be considered a weak formulation of the condition that the residual is vanishing for the true solution [Wei95]. Methods characterised by constant Z_k, i.e.

$$Z_k = Z$$

are known as generalised CG methods and correspond to methods derived from applying the CG algorithm to the normal equations and "pure" methods closely related to the basis construction algorithms of Lanczos and Arnoldi.

Note that the choice of σ_k reflects the depth to which the subspace of Eq. 2.8 is constructed and also the depth to which the orthogonality condition of Eq. 2.11 is maintained.

CG ALGORITHM

1. Initialise
   - $r_0 = b - Ax_0$
   - $p_0 = r_0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

     $\alpha_{k-1} = r_{k-1}^T r_{k-1} / p_{k-1}^T A p_{k-1}$

     $x_k = x_{k-1} + \alpha_{k-1} p_{k-1}$

     $r_k = r_{k-1} - \alpha_{k-1} A p_{k-1}$

     $\beta_k = r_k^T r_k / r_{k-1}^T r_{k-1}$

     $p_k = r_k + \beta_k p_{k-1}$

Figure 2.7: The conjugate gradient algorithm.

The Classical Conjugate Gradient Algorithm

We now discuss the classical conjugate gradient algorithm [HS52] in detail, and show how it fits into the framework presented above. Fig. 2.7 presents the algorithm, which provides an efficient means of solving linear systems of form $Ax = b$ when A is symmetric positive definite (SPD). The central idea is to minimize the function:

$$f(x_k) = \frac{1}{2} x_k^T A x_k - x^T b \tag{2.12}$$

which has a unique minimum (given SPD A) when its gradient

$$\frac{\partial f}{\partial x_k} = Ax_k - b = -r_k$$

is zero, so the value of x_k minimizing Eq. 2.12 is also the solution to $Ax = b$. To perform the function minimization, a sequence of search directions p_k are generated starting with $p_0 = r_0$; these are used to improve the iterates according to the recurrence:

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.13}$$

$$r_{k+1} = r_k - \alpha_k A p_k \tag{2.14}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k \tag{2.15}$$

where

$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$$

is chosen to minimize $f(x_{k+1})$ over the subspace $(p_0, p_1, \ldots, p_k)$ and

$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

is chosen to update the p vectors such that they are A-conjugate to one another, i.e. such that the conjugacy condition

$$p_k^T A p_j = 0 \quad \text{for } j < k$$

holds. Note that, since the p_k are non-zero and non-zero A-conjugate vectors are linearly independent, the algorithm should terminate in $m \leq n$ steps (given exact arithmetic).

Multiplying Eq. 2.13 on the left by $-A$ and adding b yields the update formula for the residuals given in Eq. 2.14. The residuals satisfy the orthogonality conditions:

$$r_k^T r_j = 0 \quad \text{and} \quad r_k^T p_j = 0 \quad \text{for } j < k \tag{2.16}$$

An inductive proof of the first orthogonality condition (i.e. $r_k^T r_j = 0$) is given in [GL89, §10.2.5]; the proof may also be derived by specialising a similar proof for the biconjugate gradient algorithm given [Fle76, §5]. The second orthogonality condition (i.e. $r_k^T p_j = 0$) follows by rewriting Eq. 2.15 as

$$p_k = r_k + \beta_{k-1} r_{k-1} + \beta_{k-1}\beta_{k-2} r_{k-2} + \ldots + (\beta_{k-1}\beta_{k-2}\ldots\beta_0) r_0 = \sum_{i=0}^{k} \gamma_i r_i \tag{2.17}$$

Changing the index from k to j and multiplying on the left by r_k yields:

$$r_k^T p_j = \sum_{i=0}^{j} \gamma_i r_k^T r_i = 0 \tag{2.18}$$

by the first orthogonality condition.

From Eq. 2.17, Eq. 2.13 can be rewritten as:

$$x_k = x_{k-1} + \alpha_{k-1} p_{k-1} = x_0 + \sum_{i=0}^{k-1} \gamma_i r_i \tag{2.19}$$

Multiplying on the left by $-A$ and adding b gives:

$$r_k = r_0 - \sum_{i=0}^{k-1} \gamma_i A r_i \tag{2.20}$$

Applying this equation to itself to yield an expression for r_k in terms of r_0 yields

$$\begin{aligned} r_k &= r_0 - \gamma_0 A r_0 - \gamma_1 A r_1 - \ldots - \gamma_{k-1} A r_{k-1} \\ &= r_0 - \gamma_0 A r_0 - \gamma_1 A (r_0 - \gamma_0 A r_0) - \gamma_2 A (r_0 - \gamma_1 A (r_0 - \gamma_0 A r_0)) - \ldots \\ &\quad - \gamma_{k-1} A (r_0 - \gamma_{k-2} A (r_0 - \ldots - \gamma_1 A (r_0 - \gamma_0 A r_0)) \ldots) \\ &= r_0 - \sum_{i=1}^{k} \delta_i A^i r_0 \end{aligned} \tag{2.21}$$

i.e. $r_k \in \text{span}(r_0, Ar_0, A^2 r_0, \ldots, A^k r_0)$, which is the Krylov subspace spanned by A and r_0. It now follows from Eq. 2.19 that:

$$x_k \in x_0 + \text{span}(r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0) \tag{2.22}$$

so the iterates lie in a shifted Krylov subspace spanned by A and r_0. This property holds for all exact conjugate Krylov subspace methods (cf. Eq. 2.10).

Note that Eq. 2.21 can be used to express r_k in polynomial form, i.e.

$$r_k = \Psi_k(A) r_0 \tag{2.23}$$

where

$$\Psi_k(A) = (I - \sum_{i=1}^{k-1} \delta_i A^i)$$

This representation is just a formal way of expressing r_k as a polynomial in A applied to a starting residual; $\Psi_k(A)$ is not explicitly computed but is rather implicitly computed as the algorithm proceeds. The importance of this residual polynomial representation will become apparent when considering the development of variants of the Bi-Conjugate Gradient algorithm such as CGS and BiCGSTAB.

Relating these results to the framework presented in the sections above, we see the CG algorithm is an exact Krylov subspace method (i.e. $\sigma_k = k$) with $q_{k-i,k} = r_{k-i}$ and $Z_k = I$. Substituting these parameters into Eq. 2.11 yields the first orthogonality condition of Eq. 2.16 i.e. $r_k^T r_j = 0$ for $k \neq j$. Also, from Eq. 2.22, the $q_{k-i,k}$ are equivalently given by $q_{k-i,k} = A^{k-i} r_0$, so $B = A$ and $z = r_0$. Since $Z_k = I$ is constant, CG falls into the class of generalised CG methods.

The convergence rate of the conjugate gradient algorithm depends on the spectral condition number $\kappa = \kappa_2(A) = \lambda_{max}/\lambda_{min}$ where λ_{max} and λ_{min} are the largest and smallest eigenvalues of A respectively [GL89, §10.2.8]. In particular, the error at iteration k is bounded by:

$$\|e\|_A = \|x - x_k\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k$$

where $\|e\|_A$ denotes the A-norm given by $\sqrt{e^T A e}$. Like optimal SOR, the rate of convergence is proportional to $\kappa^{-\frac{1}{2}}$. More complex convergence results taking into account the entire spectrum of A are given in [SV86].

From Basis Construction Algorithms to Krylov Subspace Methods

Having considered the CG algorithm in detail, we now consider the evolution of the broader class of Krylov subspace methods (c.f. Fig 2.6).

Many Krylov methods involve the construction of an orthogonal or biorthogonal basis (c.f. Eq 2.24 and Eq. 2.25) for the Krylov subspace of Eq. 2.8. Algorithms for constructing such bases have been known since the 1950s. Three of the most important algorithms, and the Krylov methods that make use of them, are described below:

- The **symmetric Lanczos algorithm** [Ste94, §4.5.1] uses short recurrences to construct an orthonormal basis $(v_1, v_2, \ldots, v_k)$ for a symmetric matrix A. At iteration k, the basis spans the Krylov subspace generated by A and v_1, i.e.

$$\mathrm{span}(v_1, v_2, \ldots, v_k) \quad = \quad \mathcal{K}_k(v_1, A) \quad = \quad \mathrm{span}(v_1, Av_1, \ldots, A^{k-1}v_1)$$

Since the generated basis is orthonormal, the vectors $v_1, v_2, \ldots, v_n$ satisfy the orthogonality condition

$$v_i^T v_j = 0 \quad \text{for } i \neq j \tag{2.24}$$

There is a close relationship between the symmetric Lanczos algorithm and the classical conjugate gradient algorithm; in fact, the conjugate gradient algorithm may be derived from the Lanczos algorithm and vice versa [GL89, §9.3.1 and §10.2.6].

An obvious way to attempt to extend the CG method to non-symmetric matrices is to multiply a non-symmetric system $Ax = b$ on both sides by A^T yielding:

$$A^T Ax = A^T b = y$$

This leads to a technique for minimizing the two-norm of the residuals (i.e. minimizing $||r||_2 = \sqrt{r_1^2 + r_2^2 + \ldots + r_n^2}$) at each step (**CGNR** [Yan94, §2.5]). Alternatively, one can solve the system:

$$AA^T z = b$$

for z and compute the desired solution as $x = A^T z$. This leads to a technique for minimizing the two-norm of the error at each step (**CGNE** [Yan94, §2.4]). However, these methods are not used in practice since they exhibit poor convergence and accuracy, have high memory requirements and require both row and column access to A.

- The **non-symmetric Lanczos algorithm** [FGN92, §3.1] generalises the symmetric Lanczos algorithm to the non-symmetric case. However, since A is non-symmetric, it is now impossible to use a single short recurrence to generate an orthonormal basis for A. Instead, the non-symmetric Lanczos algorithm uses simple three-term recurrences to construct a pair of vector sequences $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_k$ such that

$$\begin{aligned} \text{span}(v_1, v_2, \ldots, v_k) &= \mathcal{K}_k(v_1, A) \\ \text{span}(w_1, w_2, \ldots, w_k) &= \mathcal{K}_k(w_1, A^T) \end{aligned}$$

and such that the biorthogonality condition

$$w_i^T v_j = v_i^T w_j = 0 \quad \text{for } i \neq j \tag{2.25}$$

is satisfied.

Fletcher's **Bi-Conjugate Gradient (BiCG) algorithm** [Fle76] is a reformulation of the non-symmetric Lanczos algorithm. Since the residual-minimizing property of CG has been lost, BiCG can produce highly oscilliating residuals and can break down. Despite this erratic convergence behaviour and the need to perform matrix-vector multiplications with both A and A^T, BiCG is significant because it led directly to the development of several more efficient techniques with faster and/or smoother convergence.

In the BiCG algorithm, it can shown that the residual at the kth step r_k is computed as the initial residual r_0 multiplied by a matrix polynomial of degree k (c.f. Eq. 2.23). The **Conjugate Gradient Squared (CGS)** method improves on BiCG by applying this polynomial twice (like a contraction operator) to reduce r_k faster. The resulting convergence is much faster than BiCG but is sometimes more erratic, resulting in large local peaks in the convergence graphs. A further advantage of CGS is that it removes the need to multiply with A^T.

The **BiCGSTAB** method attempts to smooth the convergence of CGS by using two different polynomials to reduce r_k instead of applying the same polynomial twice. The first polynomial is the same as that used by BiCG and the second is derived from GMRES(1) (see below). **BiCGSTAB2** [Gut93a] extends the scheme to a hybrid combination of BiCG and GMRES(2) while **BiCGSTAB(l)** [SF93] takes the generalisation to its logical conclusion by combining BiCG and GMRES(l). BiCGSTAB(l) generally converges better than BiCGSTAB because it performs a better local minimization and maintains a more stable underlying BiCG process [SV95].

- **Arnoldi's method** [Ste94, §4.4.1] is another generalization of the symmetric Lanczos method to non-symmetric matrices. However, instead of constructing a biorthonormal basis for A by using short recurrences, Arnoldi's method uses long recurrences to generate a single orthonormal basis

$$(v_1, v_2, \ldots, v_k)$$

that spans the Krylov subspace generated by A and v_1 i.e.

$$\text{span}(v_1, v_2, \ldots, v_k) = \mathcal{K}_k(v_1, A)$$

The algorithm is expensive because calculation of v_k at the kth iteration requires the use of vectors $v_1, v_2, \ldots, v_{k-1}$.

The Arnoldi process was central to the development of the **Generalised Minimum Residual (GMRES) algorithm** [SS86]. The kth GMRES iterate is given by:

$$x_k = x_0 + z_k$$

where the correction z_k is chosen from the Krylov subspace

$$\mathcal{K}_k(A, r_0) = \text{span}(r_0, Ar_0, A^2 r_0, ..., A^{k-1} r_0) \tag{2.26}$$

such that z_k minimizes the two-norm of the kth residual,

$$\|r_k\|_2 = \|b - A(x_0 + z_k)\|_2 = \|r_0 - Az_k\|_2$$

Determining the correction z_k involves constructing a basis for $\mathcal{K}_k(A, r_0)$ and then solving a k-dimensional least-squares problem for the coefficients of that linear combination of the basis elements which minimizes the sum of squares of the elements of the residual vector.

GMRES is optimal in the sense that it provides the smallest residual for a fixed number of iteration steps. However, the cost of maintaining this optimality increases with each iteration step. In practical implementations, GMRES is therefore usually restarted every m iterations. This restarted form is called GMRES(m). However, for reasonable values of m (say $m \geq 20$), GMRES(m) uses so much memory that it is generally not competitive with other algorithms in terms of space.

The **QMR algorithm** of Freund and Nachtigal [FN91] also takes a least squares approach to minimizing residuals, but uses a cheaper quasi-optimal property. In addition, it uses the biorthogonal basis of BiCG which can be generated using short recurrences. The resulting algorithm requires less memory than GMRES, but still requires multiplication with A and A^T.

Since QMR is the result of applying quasi-minimal smoothing to the BiCG algorithm, it may also be beneficial to apply quasi-minimal smoothing to the CGS algorithm. Doing this leads to Freund's **Transpose-free QMR (TFQMR) algorithm** [Fre93] which, like CGS, has the advantage that it does not involve multiplications with A^T. CGS and TFQMR are closely related since TFQMR may be derived from CGS by changing only a few lines in the algorithm and the CGS iterates may be easily recovered from the TFQMR process.

For the steady-state solver of Chapter 5 we use the CGS method because it exhibits good convergence behaviour, it does not require multiplication with A^T and it has the lowest memory requirement of the algorithms for non-symmetric matrices (requiring space for A and 7 n-vectors).

Chapter 3

Contemporary Methods for Large Markov Chains

This chapter reviews several modern approaches to the problem of constructing and analysing stochastic models with large underlying state spaces. Both major phases in the analysis pipeline, i.e. *state space generation* and *steady-state solution*, are considered.

3.1 State Space Generation Techniques

3.1.1 Introduction

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

This section presents an overview of a representative cross-section of sophisticated contemporary approaches that have been developed in order to tackle this *state explosion problem.* The approaches can be classified into four categories: *probabilistic methods*, which achieve large space savings at the cost of possibly omitting states, *symbolic state space techniques* which make use of implicit state representations, *reduction techniques* which attempt to reduce the size of the individual states, the number of states in the state space and/or the number of arcs in the state graph, and *parallel and distributed approaches*, which are primarily directed at using multiple processors to reduce time and increase available memory.

3.1.2 Probabilistic Algorithms

A useful, but seemingly bizarre, method of dealing with a problem that seems to be computationally infeasible is to relax the requirement that a solution should always produce the correct answer. Adopting such a *probabilistic* or *randomized* approach can lead to dramatic memory and time savings. At the same time, the risk of producing an incorrect result needs to be quantified and kept very small if the solution is to be useful in practice.

The Case for Probabilistic Algorithms

A good example of how probabilistic algorithms can be useful arises in the primality problem, for which no polynomial time algorithm is known. The problem is to determine whether a given n-digit number x is prime. The natural solution of dividing x by every smaller number has exponential complexity (in terms of the number of digits n) and is infeasible even for moderate values of n such as 20 or 30. A slightly faster algorithm results from dividing by all prime numbers less than $\sqrt{x}$, but this is still infeasible.

Fig. 3.1 shows a probabilistic algorithm for the primality problem (adapted from [GL87]). Here k is a constant such as 20 or 30, $\mathrm{GCD}(x, y)$ is the greatest common divisor function and $J(y, x) \rightarrow (-1, 0, 1)$ is the Jacobi symbol. Both $\mathrm{GCD}(x, y)$

```
function prime(x : integer) : boolean
begin
    for n = 1 to k do begin
        y = a random number between 1 and x − 1
        if GCD(x, y) ≠ 1 then
            return false
        if y^((x−1)/2) mod x ≠ J(y, x) then
            return false
    end
return true
end
```

Figure 3.1: Probabilistic prime testing algorithm

and $J(y,x)$ are simple functions that can be calculated quickly[1].

The algorithm is based on the mathematical fact that if x is an odd prime number, then for every integer $1 < y < x-1$, it holds that $\mathrm{GCD}(x,y) = 1$ and $y^{(x-1)/2} \bmod x = \mathrm{J}(y,x)$. If x is not prime, it can be proved that there is at least a one in two chance that a randomly chosen y will violate one of these conditions [GL87]. Therefore after k loop repetitions, the chance of incorrectly reporting x as prime when it is in fact not prime, is less than 2^{-k}. By raising k we can arbitrarily increase the reliability of the algorithm at logarithmic run time cost. When k is around 40, the algorithm is probably more reliable than most computer hardware.

Besides yielding superior theoretical asymptotic run time bounds, the algorithm also performs very well in practice. Indeed, while conventional methods are limited to numbers with 30 digits or less, improved variants of this algorithm have even smaller error probabilities of 2^{-2k} and can verify the primality of 100-digit decimal numbers in under 10 minutes [Rab80].

[1] see e.g. `http://www.utm.edu/research/primes/glossary` for code for both functions

Application to State Space Generation

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

The next sections review three of the best-known probabilistic methods. In each case, we include an analysis and discussion of memory consumption and the omission probability.

Holzmann's Bit-state Hashing Method

Holzmann's pioneering bit-state hashing (or supertrace) technique [Hol91, Hol95] was developed in an attempt to maximize state coverage in the face of limited memory. The technique has proved popular because of its elegance and simplicity and has consequently been included in many research and commercial verification tools.

In Holzmann's method, the explored state table takes the form of a bit vector

T. Initially all bits in T are set to zero. States are mapped into positions in this bit vector using a hash function h, so that when state s is inserted into the table its corresponding bit $T[h(s)]$ is set to one. To check whether a state s is already in the table, the value of $T[h(s)]$ is examined. If it is zero, we know that the state has definitely not been previously encountered; otherwise it is assumed that the state has already been explored. This may be a mistake, however, since two distinct states can be hashed onto the same position in the bit vector. The result of a hash collision will be that one of the states will be incorrectly classified as explored, resulting in the omission of one or more states from the state space. Assuming a good hash function which distributes states randomly, the probability of no hash collisions p when inserting n states into a bit vector of t bits is:

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

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

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

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

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

The corresponding probability of state omission is $q = 1 - p$. Unfortunately the table sizes required to keep the probability of state omission very low are impractically large. For example, to obtain a state omission probability of 0.1% when inserting $n = 10^6$ states requires the allocation of a bit vector of 125TB. The situation can be improved a little by using two independent hash functions h_1 and h_2. When inserting a state s, both $T[h_1(s)]$ and $T[h_2(s)]$ are set to one. Likewise, we conclude s has been explored only if both $T[h_1(s)]$ and $T[h_2(s)]$ are set to one. Wolper and Leroy [WL93] show that now the probability of no hash collisions is:

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

However the table sizes required to keep the probability of state omission low are still impractically large. Using more than two hash functions helps improve the probability slightly; in fact it turns out that the optimal number of functions is about 20 [WL93]. However, computing 20 independent hash functions on every state is expensive and the resulting algorithm is very slow. The strength of Holzmann's algorithm therefore lies in the goal for which it was originally designed, i.e. the ability to maximize coverage in the face of limited memory, and not in its ability to provide complete state coverage.

Wolper and Leroy's Hash Compaction Method

Holzmann's method requires a very low ratio of states to hash table entries to provide a good probability of complete state space coverage. Consequently, a large amount of the space allocated to the bit vector will be wasted. Wolper and Leroy observed that it would be better to store which bit positions in the table are occupied instead [WL93]. This can be done by hashing states onto compressed keys of b bits. These keys can then be stored in a smaller hash table which supports a collision resolution scheme.

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

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

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

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

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

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

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

Stern and Dill's Improved Hash Compaction Method

Leroy and Wolper do not discuss exactly how states are mapped onto slots in their hash table. It seems to be implicity assumed that the hash values used to determine where to store the b-bit compressed values in the hash table are calculated using the b-bit compressed values themselves. Stern and Dill [SD95] noticed that the omission probability can be dramatically reduced in two ways – firstly by calculating the hash values and compressed values independently and secondly by using a collision resolution scheme which keeps the number of probes per insertion low. This improved technique is so effective that it requires only 5 bytes per state in situations where Wolper and Leroy's standard hash compaction requires 8 bytes per state.

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

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

1. If the slot currently being examined is empty, the compressed value is inserted into the table at that slot.

2. If the slot is occupied by a compressed value equal to the $h_3(s)$, we assume (possibly incorrectly) that the state has already been explored.

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

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

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

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

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

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

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

An upper bound for the probability of state omission q is

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

where $H_n = \sum_{k=1}^{n} 1/k$ is the nth harmonic number [SD95]. This probability rises sharply as the hash table becomes full, since compressed states being inserted are compared against many compressed values before an empty slot is found. Stern and Dill derive a more straightforward formula for the approximate maximum omission probability for a full table (i.e. with $m = n$):

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

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

3.1.3 Symbolic State Space Representations

An interesting technique for combating the state space explosion problem is to represent the state space *symbolically* instead of explicitly. The strength of this approach lies in its ability to exploit regularity and structure, resulting in compact representations of certain very large state spaces.

A popular symbolic representation that is often used in the design and analysis of digital circuits is the *binary decision diagram* (BDD) [Bry86]. Like a truth table or a Karnaugh map, a BDD is a representation of a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. Alternatively, in the context of state space exploration, a BDD can be thought of as a data structure which represents a set of bit vectors of equal length.

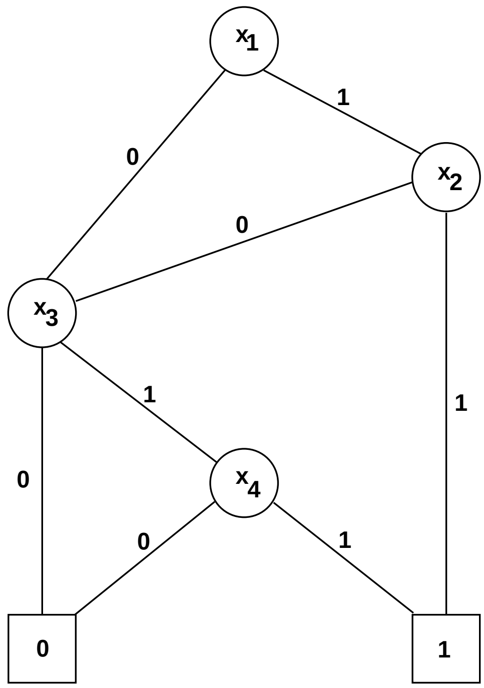


Figure 3.2: A binary decision diagram corresponding to the boolean function $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

A BDD takes the form of a directed acyclic graph (DAG) where each vertex has either zero or exactly two successors. The leaf vertices indicate the result of the function and are labelled by $\boxed{0}$ and $\boxed{1}$. Each interior vertex is labelled by a variable, and its output edges are labelled 0 and 1 corresponding to the two possible values of the variable. Fig. 3.2 shows a BDD corresponding to the boolean function $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, or, equivalently, the set $\{0011, 0111, 1011, 1100, 1101, 1110, 1111\}$. A vector $x_1x_2x_3x_4$ is in the set if and only if a traversal of the graph from the root leads to a leaf node labelled $\boxed{1}$. At each stage of the traversal, the value of x_i determines the output edge taken from node x_i.

begin
    $E = F = \{s_0\}$
    repeat
        $T = \text{succ}(F)$
        $N = T - E$
        $F = N$
        $E = E \cup N$
    until $(N = \emptyset)$
end

Figure 3.3: Algorithm for symbolic state space generation.

Fig. 3.3 represents an algorithm which performs a symbolic state space traversal starting from an initial state s_0 [PRCB94]. Here E, F, T and N are BDDs representing sets of *explored, from, to* and *new* states respectively, while $\text{succ}(F)$ is a transition function that returns a BDD describing the set of successors of the states in F.

The algorithm processes several states simultaneously, since at each step it calculates all states reachable in one step from the set of states F. Only the successor states that are new are considered in the next iteration, and the algorithm terminates when no new states are generated. The total number of iterations performed thus depends on the *depth* of the state graph, i.e. the maximum number of transitions from the initial state to the first occurrence of any of the reachable states.

BDDs have been used to analyse certain circuits with more than 10^{20} states

[BCM+92], which is many orders of magnitude beyond the bounds of known explicit state enumeration methods. However, BDDs are not always effective in preventing a state space explosion – some common circuits like combinatorial multipliers have BDDs that grow exponentially in the size of the input and certain common applications like directory-based cache coherence protocols exhibit close-to-worst-case behaviour [SD98]. In addition, small differences in the bit ordering used to encode states can have a tremendous influence on the size of the resulting BDD. However, the problem of determining the optimal encoding is known to be NP-complete [BW96]. Finally, the crucial factor in determining memory consumption of BDD-based techniques is not the size of the final BDD, but rather the peak size of the intermediate BDDs used in the state space construction algorithm. This size can be larger than the final BDD by a factor of 10 or more [PRCB94].

We conclude that BDDs are effective at representing certain very large state spaces. However, their success depends on the state space being sufficiently regular and on the application of heuristics regarding good bit orderings. Without this, BDDs do not necessarily behave better than straightforward explicit state space encodings [HMS99].

3.1.4 Reduction Techniques

Reduction techniques seek to reduce the size of the state vector, the number of states in the space space and/or the number of arcs in the state graph. Unfortunately, while many of these methods preserve logical properties, only a few can be applied in context of performance analysis unless the user is willing to accept approximate results.

Model transformations

It is often useful to apply reduction transformations to the system model before the state space is generated. Besides reducing the size of the state space and state graph, this type of pre-processing also shortens the size of the state vector. Reduction transformations are therefore usually applied before using methods that are highly sensitive to the length of the state vector, such as BDD-based approaches.

Three important types of model transformations are:

- System transformations. In some cases it is possible to use formalism-specific system transformations to reduce the model to a simpler one. Fig 3.4 shows some transformations from [Mur89] that can be applied to Petri nets, including (a) fusion of series places, (b) fusion of series transitions, (c) fusion of parallel places, (d) fusion of parallel transitions, (e) elimination of self-loop places and (f) elimination of self-loop transitions. While these transformations preserve correctness properties, they do not preserve performance analysis results.

- Redundant variable removal. The structure of models often preserves fixed relationships between certain elements in the state vector. Such relationships are often known at design time e.g. the number of customers in a closed queueing network remains constant. In addition there is a well developed theory of structural invariants to automatically derive these relationships for Petri net models [BK95]. These invariants can be exploited to safely reduce the size of the state vector by discarding superfluous elements.

- Remnant variable elimination. The future behaviour of a system usually depends on all elements of the current state vector. Sometimes, however, the evolution depends only on a subset of the elements in the state vector. This is especially the case for modular systems that consist of many components each of which has its own internal state. State vector elements outside of the subset may contain different values, but all values eventually lead to the same future behaviour. Such values are analogous to the "don't-care" values in Karnaugh maps and are known as *remnant variables* [Val98].

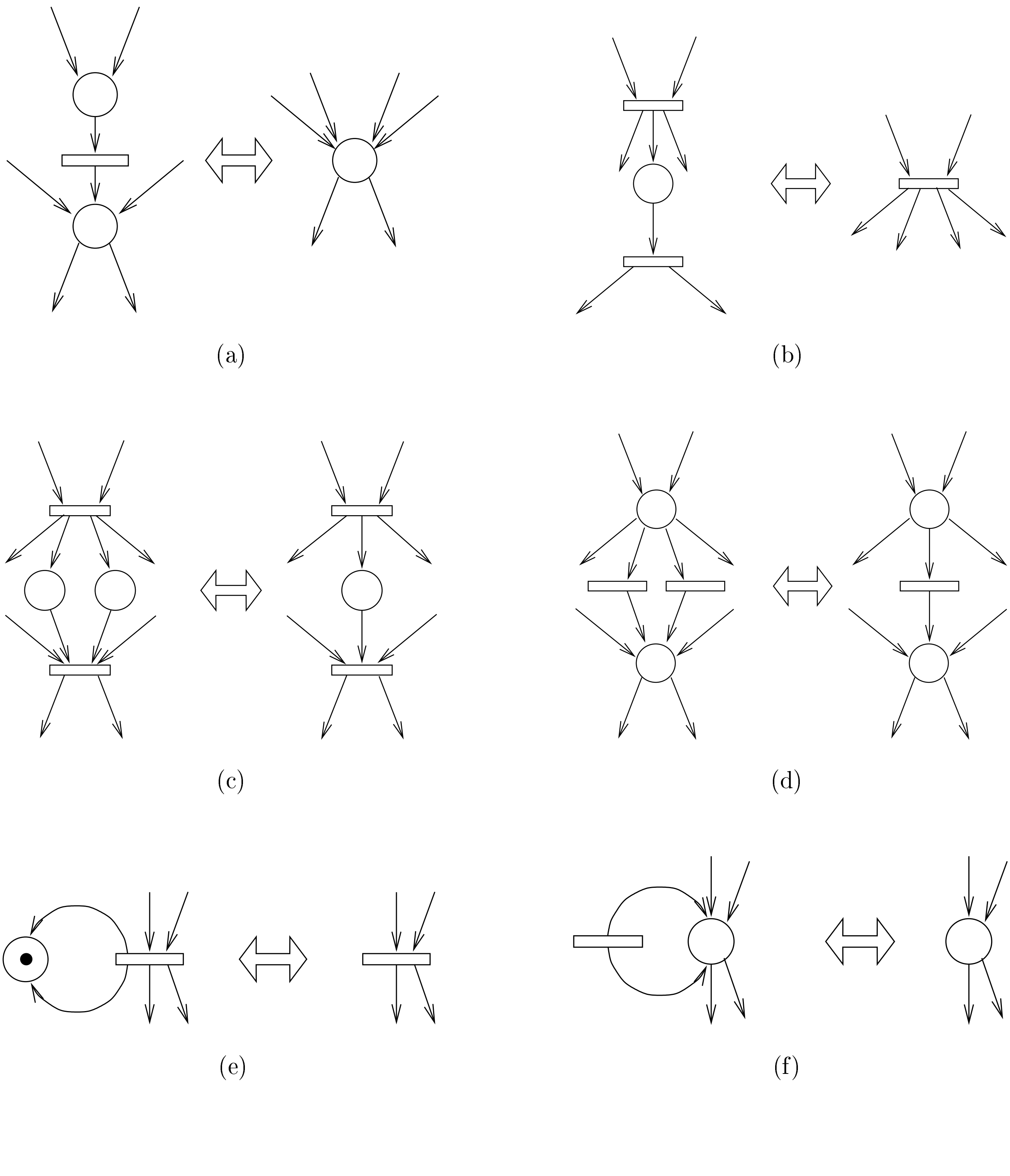


Figure 3.4: Six Petri net reduction transformations.

Remnant variables do not affect the behaviour of the system or affect the results of performance analysis, but they do contribute unnecessary states to the state space and arcs to the state graph. To eliminate these values, some verification tools support a special "don't-care" value which can be assigned to state vector elements. If an attempt is made to use a "dont-care" value, a run time error is triggered.

Partial Order Methods

One of the main causes of the state space explosion problem is the fact that all possible interleavings of concurrently enabled transitions are represented in the state graph. Partial order methods attempt to alleviate this problem by regarding concurrent executions as partial orders where concurrent *independent* transitions should be left unordered. By *independent* we mean that the order of occurrence of the transitions is irrelevant in terms of their overall effect on the state vector. The most well known partial order methods are those based on *stubborn sets* [Val91] and *persistent sets* [God96].

As an ideal application of the stubborn set method, Valmari [Val91] cites the example of n independent processes, each of which executes k steps before halting. Since the processes do not interact, the final result of the execution is independent of the order in which the system is executed. Simulating the system in only one order takes just $nk + 1$ steps instead of the $(k + 1)^n$ steps that result from considering all possible interleavings. In this way the size of the state graph can be reduced, while preserving the ability to correctly answer many analysis questions.

Of course the situation is more complicated in general, since processes do interact. In this case, several problems relating to the preservation of verification properties arise (c.f. [Val98, §7.4]), and restrictions must be placed on the method. Researchers have therefore attempted to relax the constraints on commuting the order of atomic actions without making the theory too complicated [GP93, KPV97].

While there are partial order methods that can preserve logical properties (such as boundedness, liveness or the existence deadlock), the effect of the methods on the preservation of performance measures has not been studied. Intuitively, the application of these methods to performance analysis is limited since the method destroys the structure of the underlying Markov chain by removing arcs from the state graph.

3.1.5 Parallel and Distributed Approaches

Various authors have proposed ways of tackling the high memory and time requirements of exhaustive state space exploration by using shared-memory multiprocessors or by distributing the memory requirements over several computers in a network.

Allmaier *et al.* [AH97] present a parallel shared memory algorithm for the analysis of Generalised Stochastic Petri Nets (GSPNs). 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.* [CCM95] 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.* [CGN98] 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 [CCM95] 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.

Stern and Dill parallelise their static probabilistic state enumeration method as part of the development of the Murφ correctness verifier [SD97]. The state hash table is distributed over nodes and states are assigned to nodes using a partitioning hash function. By aggregating messages sent between processors to improve efficiency, their parallel algorithm is able to achieve good speedups on a network of workstations and also on a distributed memory parallel computer. Using a network of workstations (UltraSPARC processors connected via a 38MB/s Myrinet with a low-latency active messaging system), state spaces for protocols with up to 1 200 000 states are generated over 32 processors in about 26 seconds, corresponding to a speedup of 26.6. A 63 processor IBM SP-2 parallel computer generates the same state space in 63 seconds, corresponding to a speedup of 44.2.

Finally, Stornetta [Sto95] and Ranjan *et al.* [RSBS96] present distributed memory algorithms for constructing BDDs in parallel. Stornetta uses a Meiko CS-2 parallel computer with 32 SparcStation 10 nodes (each with 32MB RAM) connected by a fat tree network, while Ranjan *et al.* use a network of 4 DEC5000 workstations (each with 40MB RAM) connected by a 70MBit/s FDDI network. In both cases, performance is poor owing to limited opportunities for parallelism and high communication cost. Speedups over the sequential version are only achieved if the sequential versions run out of physical memory and are forced to swap. However, the methods are effective at using the whole distributed memory to enable the construction of very large BDDs with up to 10 million nodes.

3.2 Steady-State Solution Techniques

3.2.1 Introduction

Having generated the state space and state graph, the second major challenge in the quantitative analysis of stochastic models is to map the state graph onto a continuous time Markov chain (CTMC) and solve it to determine the long-run proportion of time the system spends in each state. Given n states, this involves solving the (possibly very large) set of sparse linear equations

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

for π, where π is an n-vector of steady-state probabilities and Q is the $n \times n$ infinitesimal generator matrix derived from the Markov chain. Solution methods for this set of equations are limited by the large amount of memory required to store Q and π, as well as any other vectors that are used during the solution process. Another major limiting factor is the time complexity of algorithms for linear equation solution (n^3 for direct methods).

This section reviews a range of contemporary steady-state solution techniques which restrict the memory and/or time required for solution in various ways. The techniques can be classified into six categories: *Kronecker methods*, which use a memory-efficient tensor algebra representation for Q, *symbolic techniques* which use an implicit representation of the elements of Q, *"on-the-fly" methods* that construct matrix entries from the model description at solution time, *disk-based techniques* which store Q on disk and *parallel and distributed approaches*, which use multiple processors to reduce computation time and increase available memory.

3.2.2 Kronecker Methods

A limiting factor in the analysis of large Markov models is the space required to store the generator matrix Q. However, it is possible to represent Q very compactly if certain regularities in the structure of the model's underlying CTMC are ensured by the use of restricted modelling formalisms. These formalisms usually require a model to be composed of N submodels that interact in a limited way. Under this assumption, the generator matrix Q can be expressed as a set of N smaller matrices which are combined by tensor (or Kronecker) operations. This compact representation drastically reduces the memory requirements of Q. Unfortunately, for typical models, the spatial benefits do not extend to the computational complexity of sparse matrix-vector operations, which can require up to N times more computation than classical sparse matrix-vector multiplication [BCDK97].

Tensor algebra

Here we use the notation of [FPS98]. Given matrix A of dimension $(r_1 \times c_1)$ and matrix B of dimension $(r_2 \times c_2)$, the *tensor product* $C = A \otimes B$ has dimensions $(r_1 r_2 \times c_1 c_2)$ and consists of $r_1 c_1$ blocks with dimension $r_2 c_2$. Elements in C are defined by assigning the element of C in the (i_2, j_2) position in block (i_1, j_1) the value $a_{i_1 j_1} b_{i_2 j_2}$, i.e.

$$c_{\{(i_1,j_1);(i_2,j_2)\}} = a_{i_1 j_1} b_{i_2 j_2}$$

As an example, consider the matrices

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

The tensor product $C = A \otimes B$ is given by:

$$C = A \otimes B = \left(\begin{array}{ccc|ccc} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} \\ a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} \\ a_{11}b_{31} & a_{11}b_{32} & a_{11}b_{33} & a_{12}b_{31} & a_{12}b_{32} & a_{12}b_{33} \\ \hline a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} \\ a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} \\ a_{21}b_{31} & a_{21}b_{32} & a_{21}b_{33} & a_{22}b_{31} & a_{22}b_{32} & a_{22}b_{33} \end{array}\right)$$

The *tensor sum* is defined for square matrices in terms of the tensor product as:

$$A \oplus B = A \otimes I_{n_2} + I_{n_1} \otimes B$$

where n_1 is the dimension of A, n_2 is the dimension of B and I_{n_i} is the identity matrix of dimension n_i. For the example above, the tensor sum $C = A \oplus B$ is given by:

$$C = A \oplus B = \left(\begin{array}{ccc|ccc} a_{11}+b_{11} & b_{12} & b_{13} & a_{12} & 0 & 0 \\ b_{21} & a_{11}+b_{22} & b_{23} & 0 & a_{12} & 0 \\ b_{31} & b_{32} & a_{11}+b_{33} & 0 & 0 & a_{12} \\ \hline a_{21} & 0 & 0 & a_{22}+b_{11} & b_{12} & b_{13} \\ 0 & a_{21} & 0 & b_{21} & a_{22}+b_{22} & b_{23} \\ 0 & 0 & a_{21} & b_{31} & b_{32} & a_{22}+b_{33} \end{array}\right)$$

The $\otimes$ and $\oplus$ operations are associative, so that $\otimes_{k=1}^{N} A^{(k)}$ and $\oplus_{k=1}^{N} A^{(k)}$ are well defined.

Stochastic Automata Networks

Plateau first proposed the use of Kronecker operations as an efficient means of representing the generator matrices of Stochastic Automata Networks (SANs) with limited interactions [Pla85]. SANs represent collections of automata that operate largely independently, performing *local* actions, but interact infrequently to perform *synchronised* actions (rather like process algebras). A stochastic automata network consists of N components, the kth of which has n_k states.

Given N *independent* automata (executing only local actions) with generators $Q^{(1)}, Q^{(2)}, \ldots, Q^{(N)}$, the *global* generator matrix of the overall system is given by:

$$Q = \bigoplus_{k=1}^{N} Q^{(k)} = \sum_{k=1}^{N} I_{n_1} \otimes \cdots \otimes I_{n_{k-1}} \otimes Q^{(k)} \otimes I_{n_{k+1}} \otimes \cdots \otimes I_{n_N}$$

Synchronised actions may be handled by incorporating the sum of two additional tensor products for each synchronised action [FPS98]. Given E synchronising events, the global generator matrix is now given by:

$$Q = \sum_{k=1}^{2E+N} \bigotimes_{i=1}^{N} Q_j^{(i)}$$

where Q is still a generator matrix but the individual $Q_j^{(i)}$ matrices need not be. In particular, the matrices characterising synchronising transitions are often very sparse, and may consist of a single positive element.

Plateau's original approach has subsequently been extended to allow transitions to have functional rates that depend on the state of other parts of the system (provided there are no dependency cycles). In this case, the generator matrix keeps the same representation but now the $Q_j^{(i)}$ terms may contain functional elements. Generalised tensor algebra concepts supporting this extension are presented in [FPS98]. As further extensions of Plateau's work, Kronecker representations have been applied to other formalisms such as classes of queueing networks [Buc94a], stochastic Petri nets [Don94, Kem96] and stochastic process algebras [Buc94b].

Matrix-vector multiplication

Classical numerical methods such as the Power method, Jacobi's method, Gauss-Seidel and SOR, as well as more advanced Krylov subspace methods [Buc99] have all been applied to SANs. When used to solve systems represented using Kronecker algebra, these methods usally differ from their conventional counterparts only in the way they realise matrix-vector product operations.

For *dense* matrices, the product

$$x \bigotimes_{i=1}^{N} Q^{(i)}$$

where $Q^{(i)}$ is of order n_i and x is a vector of length $\prod_{i=1}^{N} n_i$, may be computed in

$$\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i$$

multiplications using the algorithm presented in [Fer98, FPS98]. This represents a reduction in the theoretical complexity of a naive matrix-vector multiplication operation which would require

$$(\prod_{i=1}^{N} n_i)^2$$

multiplications.

However, for *sparse* matrices the situation is very different. The complexity of Kronecker-based matrix-vector multiplication operations on such matrices is investigated in [BCDK97]. For the sparse $Q^{(i)}$ matrices typically encountered in practice, the computational effort required is higher than naive matrix-vector multiplication, and can increase by up to a factor of N in the worst case. The authors conclude that "the real advantage of Kronecker-based methods lies exclusively in their large memory savings".

Actual vs. potential state space

Another problem with the Kronecker approach is that it works on the *potential* state space, representing all possible interleavings of the state spaces of the component automata. This requires the allocation of solution vectors of length $\prod_{i=1}^{N} n_i$. This can result in much wasted space, since the *actual* state space may be very much smaller, depending on the extent of synchronisation. Computational effort is also wasted by calculating solution vector entries that correspond to "unreachable" states. Methods have therefore been devised to work with the actual state space. These methods require a representation of the full state space to be held in memory at solution time.

An approach developed by Ciardo and Miner based on BDD-like matrix diagrams [CM99] represents the state-of-the-art in Kronecker-based solution methods. Their method operates on the actual state space, and includes several optimisations to allow for efficient access to the matrix encoding as well as a cache to prevent duplication of floating point operations. This method is able to perform 238 Gauss-Seidel iterations on a Markov chain with 40 million states and 450 million non-zero transition matrix entries in three and a half days on a 450 MHz Pentium-II PC with 384 MB of memory. The resulting solution vector after this time is only accurate to a precision of 10^{-5}, however, since a single-precision solution vector is used to save memory. In addition, the effective throughput of transition matrix elements turns out to be only 2.5MB/s, which is very low (c.f. Section 3.2.5).

3.2.3 Symbolic Steady-state Solution

As discussed in Section 3.1.3, symbolic techniques based on binary decision diagrams (BDDs) can be used to efficiently *generate* the very large state spaces of certain structured systems. To derive performance statistics for these systems, corresponding symbolic techniques are needed in order to represent the system's underlying continuous time Markov chain, and to *solve* it for its steady-state distribution.

Compact representation of the CTMC

A symbolic data structure suitable for the representation of continuous time Markov chains is the Multi-terminal Binary Decision Diagram (MTBDD) [FMY97], also referred to as an Algebraic Decision Diagram (ADD) [BFG+97]. MTBDDs extend the range of conventional BDDs to arbitrary values, while maintaining their domain as a multi-dimensional Boolean space. That is, a MTBDD represents a function $f : \{0,1\}^n \rightarrow \mathcal{D}$ where $\mathcal{D}$ is an arbitary finite range. If, for example, $\mathcal{D}$ is the set $\{0,1\}$, the MTBDD represents a boolean function and reduces to a BDD. If $\mathcal{D}$ is a finite set of real numbers, we have a function of form $f : \{0,1\}^n \rightarrow \mathbb{R}$. This form is powerful enough to represent weighted directed graphs or, equivalently, square sparse matrices. In the context of Markov chains, this means that, given an m-bit state identifier, we can use a function

$f : \{0,1\}^{2m} \to \mathbb{R}$ to denote the transition rate between any two states in the state space.

$$R = \begin{pmatrix} 0 & 4 & 0 & 0 \\ 3 & 0 & 3 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

transition	r_1	c_1	r_2	c_2	value
$0 \xrightarrow{4} 1$	0	0	0	1	4
$1 \xrightarrow{3} 0$	0	0	1	0	3
$1 \xrightarrow{3} 2$	0	1	1	0	3
$2 \xrightarrow{2} 1$	1	0	0	1	2
$2 \xrightarrow{2} 3$	1	1	0	1	2
$3 \xrightarrow{1} 2$	1	1	1	0	1
	otherwise				0

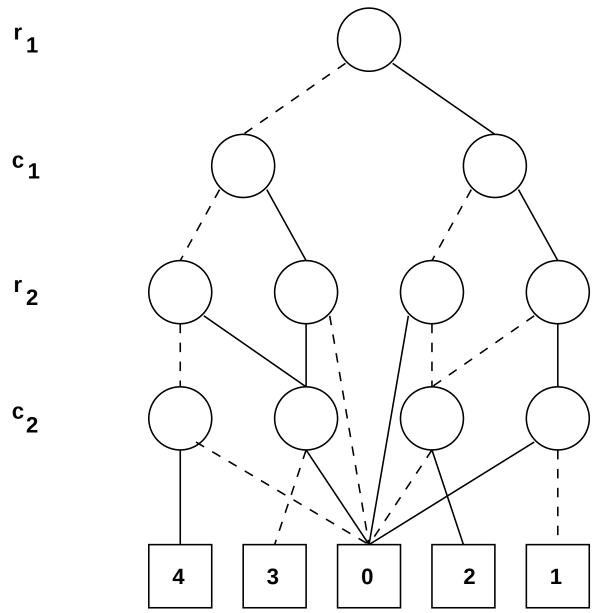


Figure 3.5: A rate matrix of a simple CTMC, its corresponding transition encoding and its MTBDD representation [HMS99].

Fig. 3.5 shows an example from [HMS99] which illustrates how the rate matrix R of a small CTMC can be encoded as an MTBDD. The matrix is 4×4 so 2 bits (r_1, r_2) are used to address its rows and 2 bits (c_1, c_2) are used to address the columns. As with BDDs, different orderings of these bits result in different MTBDDs sizes and the problem of determining the optimal ordering is NP-complete. However, a good *heuristic* is use an ordering which interleaves the row and column bits. Applying this heuristic to the example gives the ordering r_1, c_1, r_2, c_2. Fig. 3.5 shows how the entries of R are encoded using this ordering and the corresponding MTBDD representation.

The main advantages of MTBDDs over conventional sparse matrix representations is that they combine all matrix entries with the same numerical value in a single terminal vertex. In addition, given a good variable ordering, the sharing of terminal vertices can be extended to the sharing of subgraphs, particularly for matrices with a repetitive block structure. In this way, for example, the regular block-structured rate matrix of a finite tandem queue with capacity $c = 2^k - 1$ (consisting of M/Cox_2/1/c queue and an M/M/1/c queue in series) can be represented with just $30k + 3$ MTBDD vertices, even though the state space grows exponentially with k, being of size 2^{2k+1} [HMS99].

Symbolic solution

Direct methods like Gaussian elimination are not generally used for symbolic solution since they introduce changes to the structure of the coefficient matrix. These changes are expensive to reproduce in an MTBDD representation – not only is fill-in (i.e. new elements introduced when subtracting rows) likely to introduce many new terminal nodes, but considerable effort must be expended to keep the MTBDD in canonical form at every step since pivotting destroys regularity [HMPS96, BFG$^+$97]. Therefore iterative methods of form

$$\pi^{(k+1)} = \pi^{(k)} M$$

are used, where M is an MTBDD derived by applying simple operations to the transition rate matrix R. This framework allows for the implementation of the Power, Jacobi and Gauss-Seidel methods (c.f. Sec. 2.5.1), although the last involves the explicit inversion a matrix which can lead to considerable fill-in.

Fig. 3.6 shows a symbolic iterative solution algorithm (adapted from [HMS99]) for solving a system with 2^n states and an iteration matrix M to an accuracy of

function iterative-solve(M, n, ϵ)
begin
  $P = 1/2^n$
  repeat
    $P' =$ vector-matrix-multiply(P,M)
    $T =$ max-abs(apply($P, P', -$))
    $P = P'$
  until value(T) $< \epsilon$
  return P
end

Figure 3.6: Algorithm for iterative symbolic steady-state solution.

ϵ . Both the steady-state solution vector P and the iteration matrix M are represented as MTBDDs. The solution vector is initialised to be an MTBDD with a single node with value $1/2^n$ so that all states are assumed to be equiprobable. During each iteration, P and M are multiplied using an MTBDD vector-matrix multiplication operation to produce the BDD P'. This operation has the same complexity as conventional sparse matrix-vector multiplication, with the added advantage that, like other operations which combine BDDs, the procedure can make use of sophisticated hashing techniques to avoid recalculations of intermediate results (see e.g. [Bry86]). However, extra overhead is needed to maintain the MTBDD in reduced canoncial form.

The remaining steps of the algorithm compute the convergence criterion

$$||\pi^{(k+1)} - \pi^{(k)}||_\infty < \epsilon$$

by applying the subtraction operator over P and P' and using the max-abs operator to determine the largest absolute value. If this value is less than ϵ, the algorithm terminates and returns the steady-state vector π represented by the MTBDD P.

Implementations of MTBDD solution algorithms have been able to give approximate solutions for the steady-state distributions of very large Markov chains with up to 10^{27} states in only 9 iterations requiring just 28 seconds and 71MB of memory [HMPS96]. The results are approximate because small probabilities in the solution vector have to be rounded to zero during computation in order to prevent the number of distinct probabilities values in the solution vector from growing too large.

The chains in [HMPS96] were derived from finite state machines representing digital circuits. Progress on applying MTBDDs to the solution of chains derived from more complex formalisms (such as GSPNs and SPAs) has been slower, since these chains generally have a more complex CTMC structure with less regularity and more varied steady-state solution distributions with many distinct values. The recent results showing that certain structured CTMCs which exhibit exponential growth in the number of states can be represented with MTBDDs that grow linearly in the number of nodes is a promising first step, although these large models have yet to be solved. Further work is also needed to find efficient symbolic variants of more powerful numerical techniques (such as Krylov subspace techniques).

3.2.4 On-the-fly Methods

"On-the-fly" solution techniques [DS98b] seek to tolerate large state spaces by avoiding explicit storage of the generator matrix. The idea is to derive generator matrix entries as they are needed during each iteration using only the state space, model description and transition rules. The advantage of this method is that it can be applied to unrestricted models, and that no space (either on disk or in memory) is required to store the transition matrix. However, the method is very slow and memory must still be reserved to store the state space, in addition to any vectors needed by the solution process.

Rows of the generator matrix can be generated on-the-fly in a simple way. The state space is held in memory, either using a tree data structure or a hash table and each state is associated with a unique index. The ith row of Q can then be computed by: (a) applying a successor function to the ith state s_i in order to determine the tangible successor states and the rates of the corresponding transitions and (b) searching the tree or hash table to determine the index of each succesor state.

Generating rows of Q is therefore straightforward, using similar functions and lookups as those required during a full-scale state space exploration. However, this limits the choice of numerical method to the Jacobi or Power methods which typically converge slowly. Access to the columns of Q (i.e. the rows of Q^T) is required to implement more powerful techniques like Gauss-Seidel (when applied to the system $Q^T\pi^T = 0$). Finding the entries in column i of Q involves finding the set of predecessor states of state s_i and corresponding incoming rates. This can be done by executing the model "backwards" in time. For each Gauss-Seidel step the model also needs to be executed "forwards" in order to determine the diagonal element q_{ii} (unless of course extra memory is reserved to store these elements).

Executing models backwards turns out to be a simple operation for SPNs. By constructing a *reverse model*, i.e. a model in which all arcs are reversed and where any marking-dependent rates are determined after transition firing, the set of predecessor states can be quickly determined. The procedure is slightly more complicated for GSPNs since it is necessary to recursively search through networks of immediate transitions to determine the set of tangible predecessors.

For more complicated formalisms like Stochastic Activity Networks and Stochastic Reward networks, it is "virtually impossible to define a reverse model in closed or simple algorithmic form" [DS98b]. However, if the models are sparsely connected and have small bounds on the number of tokens in places, it is possible to use an algorithm which conducts an exhaustive search of all possible predecessors.

Since generating rows and columns are expensive, on-the-fly techniques are best used with numerical techniques that exploit locality, such as Block Gauss-Seidel. The idea is to generate and then cache matrix blocks in memory, using them several times within an iteration. On every *outer iteration*, Block Gauss-Seidel uses off-diagonal matrix blocks once to calculate:

$$\hat{r}_i = \sum_{j=1, j \neq i}^{j=N} Q_{ij}^T \hat{\pi}_j$$

where Q_{ij}^T is a submatrix of Q^T and $\hat{\pi}_j$ is a subvector of the steady-state vector π. Then several *inner iterations* are repeatedly performed to solve the diagonal system:

$$Q_{ii}^T \hat{\pi}_i = \hat{r}_i$$

The more inner iterations that are performed, the better reuse is made of matrix blocks and the fewer outer iterations are required. However, diminishing returns are observed so typically the number of inner iterations has to be tuned to some optimum for each model.

Deavours and Sanders further present a Block Gauss-Seidel technique that requires only row access to the matrix Q at the cost of an extra solution vector. Running on a 160MHz HP C-160 workstation, their method requires 39 762 seconds (11 hours) and 57MB of memory to solve a model of a Kanban manfacturing system with 2 546 432 states and 24 460 416 non-zero entries in the generator matrix to an accuracy of 10^{-6}.

In summary, on-the-fly solution techniques provide a way of solving unrestricted models expressed in several different formalisms without explicitly storing the transition matrix. However, the method is slow and memory is still required to store the state space S as well as any vectors needed by the solution process.

3.2.5 Disk-based Solution Techniques

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

In [DS97] and [DS98a], Deavours and Sanders make a compelling case for the potential of disk-based steady-state solution methods for large Markov models. They note that our ability to solve large matrices is limited by the memory required to store a representation of the transition matrix and by the effective rate at which matrix elements can be produced from the encoding. As a general rule, the more compact the representation, the more CPU overhead is involved in retrieving matrix elements. Two common encodings are Kronecker representations and "on-the-fly" methods. Deavours and Sanders estimate the effective data production rate of Kronecker and "on-the-fly" methods as being 2 MB/s and 440 KB/s respectively on their 120 MHz HP C110 workstation. Recently published results show that an implementation of a state-of-the-art Kronecker technique running on a 450 MHz Pentium-II workstation yields an effective data production rate of around 2.5 MB/s [CM99].

At the same time, modern workstation disks are capable of sustaining data transfer rates of 5–10 MB/s (and even higher rates are possible if disks are interleaved). This suggests that it would be worthwhile to store the transition matrix on disk, given that enough disk space is available and given that we can apply an iterative solution method that accesses the transition matrix in a predictable way. Such an approach has the potential to produce data faster than both Kronecker and on-the-fly methods, without any of the structural restrictions inherent in Kronecker methods.

Deavours and Sanders demonstrate the effectiveness of this approach by devising a sequential disk-based solution tool which makes use of two cooperating processes. One of the processes is dedicated to reading disk data while the other performs computation using a Block Gauss-Seidel algorithm, thus allowing for the overlap of disk I/O and computation. The processes communicate using semaphores and shared memory. As outlined in Sec. 3.2.4, the advantage of using Block Gauss-Seidel is that diagonal matrix blocks can be read from disk once, be cached in memory and then reused several times.

The memory required by the disk-based approach is small – besides the shared memory buffers, space is only required for the solution vector itself. This enables the solution of extremely large models with over 10 million states and 100 million non-zero entries on a HP C110 workstation with 128MB RAM and 4GB of disk space in just over 5 hours. These results confirm that disk-based methods are the method of choice for solving large Markov models, provided enough disk space is available to hold the transition matrix.

3.2.6 Parallel and Distributed Approaches

As was the case for state generation, several authors have tried to overcome the high memory and time requirements of steady-state solution by using parallel and distributed solution techniques.

In [MCC97], Caselli *et al.* extend their work concerning distributed state space generation for GSPN models [CCM95] to include a distributed solution phase. The solution process begins by distributing the transpose of the generator matrix Q^T across the processors in a row-wise fashion. As well as a portion of transition matrix, each processor stores a full copy of the solution vector. This reduces the communication load, at the cost of limiting the scalability of the technique. Iterations are performed using a modified parallel Gauss-Seidel algorithm whereby each processor applies a standard Gauss-Seidel iteration to its portion of the matrix, using its private copy of the solution vector. No communication takes place during an iteration. However, at the end of an iteration, only one processor broadcasts its set of updated solutions. The processors take turns so that, given N processors, the broadcast after iteration i is performed by processor $i \bmod N$. This reduces communication time further, but the convergence rate of the algorithm is adversely affected. Using a network of four 75 MHz Pentium processors with 32 MB RAM and connected by 100 Mb/s Ethernet, Caselli *et al.* are able to solve systems of up to 590 000 states in about 10 minutes. The speedup over sequential solution varies depending on the model – for one example there is a slight slowdown while for another there is a speedup of about 2 (i.e. 50% efficiency).

In [AKH97], Allmaier *et al.* consider the solution of GSPN models on a shared memory multiprocessor using Horton's multi-level algorithm [HL94]. The multi-level algorithm uses aggregation-disaggregation type steps to solve recursively coarsened representations of the original Markov chain. The algorithm has two phases, viz. the *aggregation* phase and the *iteration* phase. During the aggregation phase, a hierarchical system of coarsened CTMCs (called levels) is set up. The smallest of these levels is small enough to be solved quickly and exactly using a direct method. During the iteration phase, some smoothing Gauss-Seidel iterations are performed on the current level, transition probabilities in the next coarser level are updated, and an iteration is recursively conducted on the next coarser level. At the coarsest level, the CTMC is solved exactly to provide improved values for the probabilities of aggregates. These improved values can then be propagated backwards to refine the lower-level probabilities. While the *aggregation* phase is largely sequential and difficult to parallellise, the *iteration* phase parallelises well if multiple threads are used. Using a Convex SP1600 shared-memory multiprocessor with 4 GB of main memory, a CTMC with 190 000 states and 1 400 000 arcs is solved in about 125 seconds on 8 processors. The corresponding speedup over sequential solution is 3.4 (i.e. an efficiency of 42.5%).

Recently, Buchholz *et al.* reported results on the parallelization of Kronecker methods using a cluster of workstations [BFK99]. Since the Kronecker representation of the generator matrix is very compact, distribution of the matrix poses no problems – the full Kronecker structure can be rapidly broadcast to every processor. Since the Kronecker representation used is a hierarchical one that partitions naturally into a block structure, submatrices are then assigned to processors. For each $n_x \times n_y$ block assigned to a processor, 3 vectors of length n_x are required; this includes a portion of the global steady-state vector. Each processor also needs one longer vector to store intermediate results. The solution algorithm itself uses a master-slave architecture; a master process manages issues such as normalisation and convergence monitoring, while several worker processes implement a parallel two-level block Jacobi algorithm (with relaxation). As with other block solution methods, there is a delicate balance between the number of inner and outer iterations required to achieve optimum convergence. To save communication during outer iterations, updates to steady-state vector elements are sent between processors only if there is a relatively large difference between successive local iterates. Results are reported for a network of Sun workstations connected by 100 Mbps Ethernet. For a flexible manufacturing system model (c.f. Appendix A.2) with 1.6 million states and 14 million non-zero off-diagonal entries in Q, the time for solution is 4 890 secs using a dedicated master processor and a network of 4 UltraSPARC worker processors, each with 192MB and a 270–300MHz CPU. The corresponding speedup is 1.70 (i.e. 42.5% efficiency). For a larger model of a multiclass queueing system with 8 million states and 51 million non-zero off-diagonal entries in Q, the time taken using 6 UltraSPARC worker processors (167-300MHz) is 6 990 secs. This corresponds to a speedup of 1.78 (29.7% efficiency). In both cases, the accuracy of the solution is such that the infinity norm of the normalised residual (given by $||r||_\infty = \max_i |r_i|$) is less than 10^{-9}.

Chapter 4

Probabilistic Dynamic Distributed State Space Generation

4.1 Introduction

This chapter describes a novel parallel state space generation algorithm based on a dynamic hash compaction storage scheme. To make this chapter self-contained, as in the publication [KHMK99], we begin by summarising relevant background material from Chapters 2 and 3 before giving full details of the new method.

Complex systems can be modelled using high-level formalisms such as stochastic Petri nets, queueing networks 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 graph*. The state space consists of all reachable states or configurations that the system can enter starting from some initial state, while the state graph describes transitions between states. By examining the state space and state graph it is possible to detect transition sequences that lead to unsafe states or undesirable situations such as deadlock. Further, given the rate of movement between states, the state graph can be mapped on a Markov chain which can be solved to obtain performance statistics.

In certain cases, where the state space has sufficient structure, it is possible to obtain an efficient analytical solution 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 [BCMP75]. Further, certain restricted hierarchical structures allow states to be aggregated and the state space to be decomposed [Buc95, Kem96]. Here, however, we consider the general problem where no symmetry or other structure is assumed.

Conventional methods for state space generation have high memory requirements and are computationally intensive, which makes them unsuitable for generating the very large state spaces of real-world systems. Two of the most promising approaches that researchers have developed over the past decade in an attempt to resolve this problem for structurally unrestricted state spaces are:

- **Probabilistic methods:** A probabilistic algorithm is one which uses randomization techniques in an attempt to reduce the time and/or space complexity of a problem. This improvement comes at a cost, however, since there is a non-zero probability that the algorithm will produce an incorrect answer. In the context of state exploration algorithms, probabilistic techniques based on hash compaction can result in dramatic memory savings if users are prepared to tolerate a risk of state omission. The best known probabilistic state space generation algorithms are described in Section 3.1.2.

- **Parallel and distributed techniques:** With the availability of distributed-memory computers and high-speed workstation clusters, much attention has been focused on reducing processing time using multiple processors. The work carried out so far is described in Section 3.1.5 and has shown that a good load balancing scheme and a careful analysis of communication cost are important to achieve good speedups.

This chapter presents a new distributed probabilistic state exploration algorithm which combines these two approaches. The technique, which is based on a distributed dynamic hash compaction storage scheme, has several important advantages:

- Memory consumption is low and independent of the number of elements in the state descriptor.
- Access to states is simple and rapid.
- The algorithm has a low state omission probability which compares favourably with the best probabilistic methods. Furthermore, the chance of state omission may be arbitrarily reduced by performing multiple runs with independent sets of hash functions.
- Unlike existing probabilistic methods which are based on the static preallocation of large blocks of memory, our algorithm uses dynamic memory allocation, which ensures memory is only allocated as needed.
- The algorithm delivers good speedups and scalability.

The remainder of this chapter is organised as follows. After introducing sequential state space exploration in Section 4.2, we give details of the dynamic storage allocation scheme in Section 4.3 and of the parallel state space generation algorithm in Section 4.4. An enhanced communication-efficient version of the parallel algorithm is presented in Section 4.5. A theoretical performance model for both versions of the parallel algorithm is developed in Section 4.6 and numerical results demonstrating the observed performance of the algorithm on a Fujitsu AP3000 parallel computer are given in Section 4.7. Section 4.8 discusses suitable hashing and partitioning functions and Section 4.9 concludes and considers further work.

4.2 Sequential State Space Exploration

The goal of state space exploration is to map a high-level model description onto its underlying low-level state space and state graph. Fig. 4.1 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. Also, as we will see in Chapter 5, using a breadth-first approach induces a structure on the generator matrix that can be exploited during the solution process.

```
begin
    E = {s0}
    F.push(s0)
    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
```

Figure 4.1: Simple sequential state space generation algorithm

The function succ(s) in Fig. 4.1 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 "instantaneous 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 [CMT91, Kno96], so that succ(s) returns a set of non-vanishing or *tangible* successor states. The case where s_0 is vanishing is not considered in Fig. 4.1, but can be handled by initially inserting each $s \in \text{succ}(s_0)$ into F and setting $E = \text{succ}(s_0)$.

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 $\text{id}(s)$. 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 to disk row by row as the algorithm proceeds, so there is no need to store it in main memory.

4.3 Dynamic Probabilistic Hash Compaction

The memory consumed by the state exploration process depends on the layout and management of the two main data structures of Fig. 4.1. The FIFO queue F 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 held on disk. The table of explored states E, on the other hand, enjoys no such locality of access, and 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 use hashing techniques to drastically reduce the memory required to store states. However, 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 it can be quantified and kept very small.

Some of the best known probabilistic techniques are Holzmann's bit-state hashing method [Hol91, Hol95], Wolper and Leroy's hash compaction technique [WL93] and Stern and Dill's enhanced hash compaction method [SD95] (c.f. Section 3.1.2). All these methods rely on *static* memory allocation since they pre-allocate large blocks of memory for the explored state table. Since the number of states is 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 probability. The system is illustrated in Fig. 4.2. 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 and state sequence numbers. The state sequence numbers act as unique identity tags, thus enabling the efficient construction of the state graph.

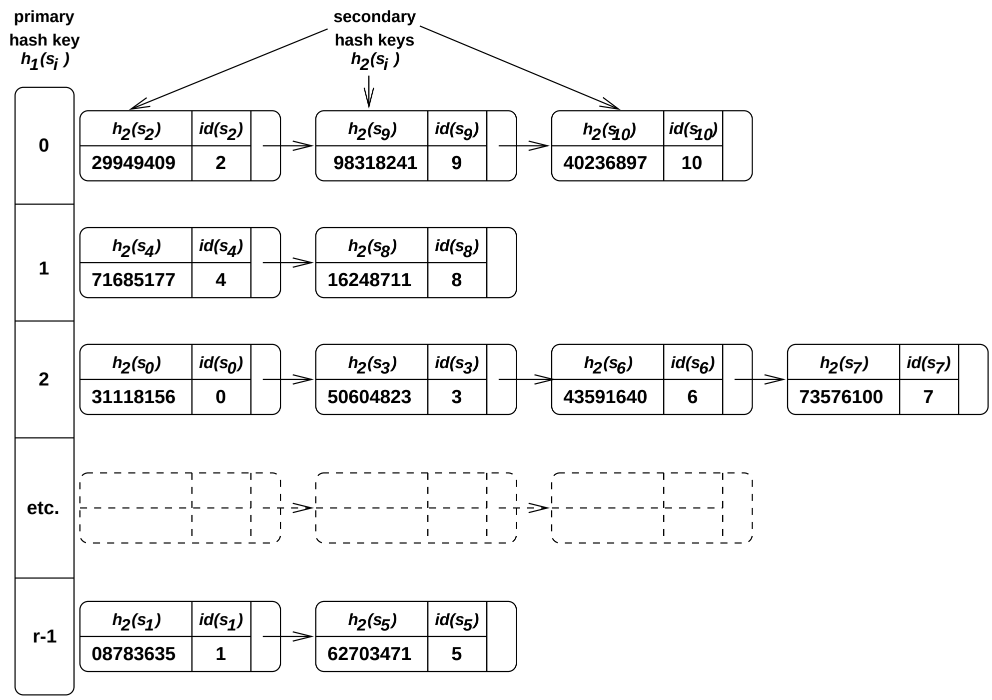


Figure 4.2: Layout of the explored state table under the dynamic probabilistic hash compaction scheme

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

begin
    $H = \{[h_1(s_0), h_2(s_0)]\}$
    $F.\mathrm{push}(s_0)$
    $E = \{\ s_0\ \}$
    $A = \emptyset$
    while (F not empty) **do begin**
        $F.\mathrm{pop}(s)$
        for each $s' \in \mathrm{succ}(s)$ **do begin**
            if $[h_1(s'), h_2(s')] \notin H$ **do begin**
                $F.\mathrm{push}(s')$
                $E = E \cup \{s'\}$
                $H = H \cup \{[h_1(s'), h_2(s')]\}$
            end
            $A = A \cup \{\mathrm{id}(s) \to \mathrm{id}(s')\}$
        end
    end
end

Figure 4.3: Sequential dynamic probabilistic state space generation algorithm

Fig. 4.3 shows the complete sequential dynamic probabilistic state space generation algorithm based on our hash compaction technique. Here H represents the state hash table of Fig. 4.2. Each state $s \in E$ has an entry of form $[h_1(s), h_2(s)]$ in H. Since it is now not necessary to store the full state space E in memory, the insertion of states into E can be handled by writing the states to a disk file as they are encountered. These states will only be required again during the calculation of performance measures after the steady-state solution phase.

Note that two states s_1 and s_2 are classified as being equal if and only if $h_1(s_1) = h_1(s_2)$ and $h_2(s_1) = h_2(s_2)$. This may happen even when the two states are different, so collisions may occur (as in all other probabilistic methods). However, as we will see in the next section, 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 different sets of independent hash functions and comparing the resulting number of states and transitions, it is possible to further arbitrarily decrease the risk of an undetected collision.

4.3.1 Reliability of the Probabilistic Dynamic State Hash Table

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

$$p = \frac{(rt)!}{(rt-n)!(rt)^n} \tag{4.1}$$

An equivalent formulation of Eq. 4.1 is:

$$p = \prod_{i=0}^{n-1} \frac{rt-i}{rt} = \prod_{i=0}^{n-1} \left(1 - \frac{i}{rt}\right) \tag{4.2}$$

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

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

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

$$p \approx e^{-\frac{n^2}{2rt}} \tag{4.3}$$

We now show that if $n^2 << rt$ then this approximation is also a lower bound for p (and thus provides a conservative estimate for the probability of complete state coverage).

First we note that, for $|x| << 1$ (in fact for any $x < 0.683803$),

$$\begin{aligned}
e^{-(x+x^2)} &= 1-(x+x^2)+\frac{(x+x^2)^2}{2!}-\frac{(x+x^2)^3}{3!}+\dots \\
&= 1-x-x^2/2+5x^3/6+\dots \\
&\leq 1-x
\end{aligned}$$

Using this fact with $x = -i/rt$ in Eq. 4.2 yields:

$$\begin{aligned}
p &= \prod_{i=0}^{n-1}\left(1-\frac{i}{rt}\right) \\
&\geq \prod_{i=0}^{n-1} e^{-(i/rt+i^2/(rt)^2)} \\
&= e^{\sum_{i=0}^{n-1} -(i/rt+i^2/(rt)^2)} \\
&= e^{-\frac{n(n-1)}{2rt}-\frac{n(n-1)(2n-1)}{6(rt)^2}} \\
&= \left(e^{-\frac{n^2}{2rt}}\right)\left(e^{\frac{n}{2rt}-\frac{n(n-1)(2n-1)}{6(rt)^2}}\right) \\
&= \left(e^{-\frac{n^2}{2rt}}\right)\left(e^{\frac{n(3rt-(n-1)(2n-1))}{6(rt)^2}}\right)
\end{aligned} \tag{4.4}$$

Our approximation of Eq. 4.3 will be a safe lower bound for p if the term

$$e^{\frac{n(3rt-(n-1)(2n-1))}{6(rt)^2}}$$

is greater than 1 (since then the approximation of Eq. 4.3 produces an omission probability *less* than the lower bound of Eq. 4.4). Since $e^x > 1$ for $x > 0$, the term above will be greater than 1 when:

$$3rt-(n-1)(2n-1) > 0$$

Our scheme is only useful if $n^2 << rt$ (so that p is near 1), in which case the above condition holds. Thus the approximation of Eq. 4.3 is also a lower bound for p for all cases of practical interest.

Assuming $n^2 << rt$, we can again use the fact that $e^x \approx (1+x)$ for $|x| << 1$ in Eq. 4.3 to approximate p by:

$$p \approx 1 - \frac{n^2}{2rt} \tag{4.5}$$

Note that, since $e^{-x} \geq 1 - x$, we have

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

so the simple formula of Eq. 4.5 is also a lower bound for p.

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

$$q = 1 - p \leq \frac{n^2}{2rt} = \frac{n^2}{r2^{b+1}}. \tag{4.6}$$

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.

4.3.2 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:

$$M = hr + n(b+s)/8. \tag{4.7}$$

Here h is the number of bytes of overhead per hash table row and s is the number of bits required to store a state sequence number. 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. 4.6:

$$r \approx \frac{n^2}{q2^{b+1}} \tag{4.8}$$

q	number of states								
	10^6			10^7			10^8		
	MB	b	r	MB	b	r	MB	b	r
0.001	8.284	34	29 104	86.87	38	181 899	908.9	41	2 273 737
0.01	7.872	31	23 283	82.84	34	291 038	868.7	38	1 818 989
0.1	7.470	28	18 626	78.72	31	232 831	828.4	34	2 910 383

Table 4.1: 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

and substituting this into Eq. 4.7 yields

$$M \approx \frac{hn^2}{q2^{b+1}} + \frac{n(b+s)}{8}$$

Minimizing M with respect to b gives:

$$\frac{\partial M}{\partial b} \approx -\frac{n^2(\ln 2)h}{q2^{b+1}} + n/8 = 0$$

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

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

The corresponding optimal value of r can then be obtained by substituting b into Eq. 4.8.

Table 4.1 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 state sequence number size of $s = 32$ bits and a hash table row overhead of $h = 8$ bytes per row. The latter corresponds to a straightforward dynamic array implementation that uses a 32-bit pointer to the start of the array and a 32-bit word for the number of elements in the array. Note that 16 bits would also suffice to store the number of elements in the array since, in the optimal case, each hash row only contains an average of 30 to 50 entries and the variance of the number of states allocated to each hash table row is low (assuming a good hash function which distributes states according to a binomial distribution). 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.

4.3.3 Comparison with Contemporary Probabilistic Methods

We now compare the reliability and space complexity of our dynamic storage scheme with the contemporary probabilistic methods described in Section 3.1.2. For Holzmann's method, the probability of state omission can be approximated by

$$q \approx \frac{n^2}{2l}$$

where l is the length in bits of a large pre-allocated bit vector. However, to obtain a low omission probability the ratio of states to bit vector entries must be kept very small. Consequently, a large amount of the memory allocated for the bit vector will be wasted. Our method solves this problem by providing an extremely large "virtual" bit vector of size $l = rt = r2^b$ without physically allocating memory space for the vector. Instead, our scheme effectively stores only the positions in the bit vector that are occupied, resulting in dramatic memory savings.

For Wolper and Leroy's method, the state omission probability is

$$q \approx \frac{n^2}{2t}$$

where $t = 2^b$ and b is the number of bits used to store compressed states in a large pre-allocated hash table. Our method improves on this probability by a factor of r in the denominator. This is because each of our r hash table rows effectively corresponds to a separate Wolper and Leroy hash table, and states are assigned to the hash table rows using a primary hash function that is *independent* of the secondary hash function used to compress states. Because each hash table row contains only a few states, our method requires fewer bits per compressed value to store states with the same omission probability.

Stern and Dill's method has a complicated omission probability that depends on the number of free slots in the explored state table. Given a hash table with m

slots, n of which are used,

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

where again b is the number of bits used to store compressed states in a large pre-allocated hash table. For a full or nearly full hash table, this omission probability rises rapidly since an insertion operation will involve comparisons with many compressed values before a free slot is found. Our method does not have this problem since the end of each row in our hash table behaves in the same way as a free slot in a Stern and Dill hash table, thus limiting the number of comparisons that need to be made when inserting.

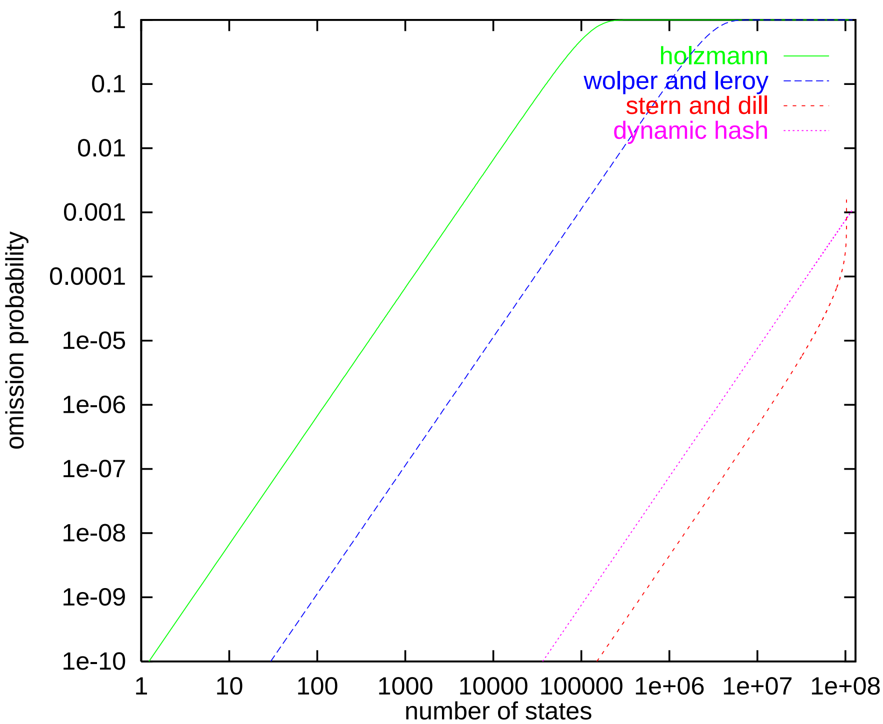


Figure 4.4: Contemporary static probabilistic methods compared with the dynamic hash compaction method in terms of omission probability

Fig. 4.4 compares the omission probability of contemporary static probabilistic methods with that of our dynamic hash compaction method for state space sizes of various magnitudes up to 10^8. The parameters used for each method are presented in Table 4.2, and are selected such that the memory use of all four algorithms is the same. The graph shows that our method yields a far lower omission

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

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

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

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

Table 4.2: Parameters used in the comparison of omission probabilities

probability than both Holzmann's method and Wolper and Leroy's method. In addition, our algorithm is competitive with Stern and Dill's method and yields a better omission probability when the hash table becomes full or nearly full.

4.4 Parallel Dynamic Probabilistic State Space Exploration

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

In the parallel algorithm, the state space is partitioned between the nodes so that each node is responsible for exploring a portion of the state space and for constructing part of the state graph. A partitioning hash function $h_0(s) \rightarrow$

$(0, \ldots, N-1)$ is used to assign states to nodes, such that node i is responsible for exploring the set of states E_i and for constructing the portion of the state graph A_i where:

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

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. Guidelines for choosing hash functions which meet these goals will be discussed in Section 4.8.

The operation of node i in the parallel algorithm is shown in Fig. 4.5. Each node i has a local FIFO queue F_i used to hold unexplored local states and a hash table H_i representing a compressed version of the set E_i, i.e. those states which have been explored locally. State s is assigned to processor $h_0(s)$, which stores the state's compressed state descriptor $h_2(s)$ in the local hash table row given by $h_1(s)$. As before, it is not necessary to store the complete state space E_i in memory, since states can be written out to a disk file as they are encountered.

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

```
begin
    if h_0(s_0) = i do begin
        H_i = {[h_1(s_0), h_2(s_0)]}
        F_i.push(s_0)
        E_i = {s_0}
    end else
        H_i = E_i = ∅
    A_i = ∅
    while (shutdown signal not received) do begin
        if (F_i not empty) do begin
            s = F_i.pop()
            for each s' ∈ succ(s) do begin
                if h_0(s') = i do begin
                    if [h_1(s'), h_2(s')] ∉ H_i do begin
                        H_i = H_i ∪ {[h_1(s'), h_2(s')]}
                        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 [h_1(s'), h_2(s')] ∉ H_i do begin
                H_i = H_i ∪ {h_1(s'), h_2(s')}
                F_i.push(s')
                E_i = E_i ∪ {s'}
            end
            send-id(k, g, id(s'))
        end
    end
end
```

Figure 4.5: Parallel state space generation algorithm for node i

the state identity numbers of non-local successor states. Therefore, when a node receives a state it returns its identity to the sender by calling send-id(k, g, h) where k is the sender, g is the identity of the parent state and h is the identity of the received state. The identity is received by the original sender via a call to receive-id(g, h). Fig. 4.6 summarises the main steps that take place to identify and process each child s' of state s in the case that $h_0(s) \neq h_0(s')$.

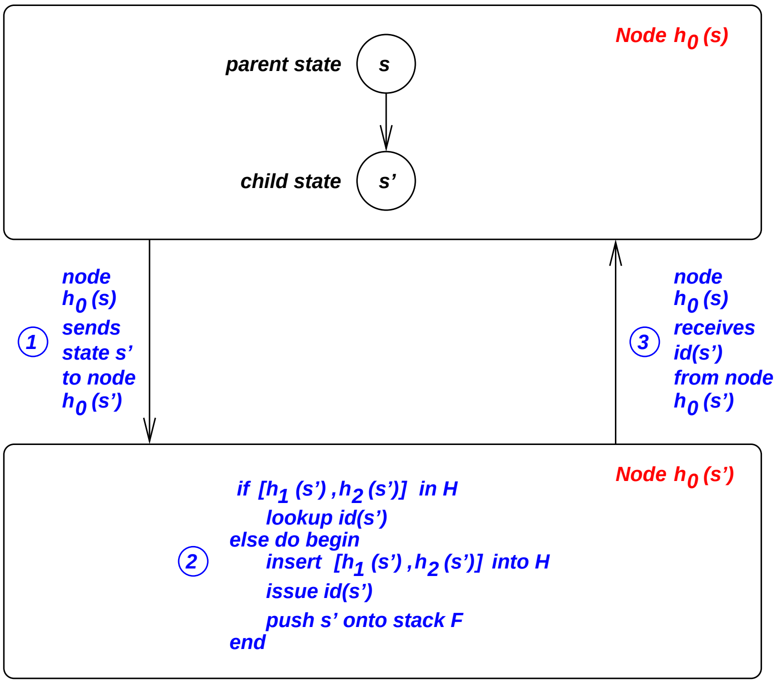


Figure 4.6: Steps required to identify child state s' of parent s.

Given a state graph consisting of a arcs and n states, and assuming a uniform distribution of states across the N processors, the total communication cost (in bytes) of our algorithm across all nodes is:

$$C(a, N) = \frac{(N-1)}{N} a(C_i + C_s)$$

where C_i and C_s reflect the cost of sending a state identity and a state descriptor respectively.

In practice, it is inefficient to implement the communication as detailed in Fig. 4.5 and Fig. 4.6, since the network rapidly becomes overloaded with too many short messages. Consequently state and identity messages are buffered and sent in large blocks. In order to avoid starvation and deadlock, nodes that have very few states left in their FIFO queue or are idle broadcast a message to other nodes requesting them to flush their outgoing message buffers.

The algorithm terminates when all the F_i's are empty and there are no outstanding state or identity messages. The problem of determining when these conditions are satisfied across a distributed set of processes is a non-trivial problem. From the several distributed termination algorithms surveyed in [Ray88], we have chosen to use Dijkstra's circulating probe algorithm [DFG83].

4.4.1 Reliability of the Parallel Probabilistic Algorithm

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

$$q = \frac{n^2}{Nr2^{b+1}}. \tag{4.9}$$

4.4.2 Space Complexity

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

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

4.5 An Enhanced Communication-efficient Parallel Algorithm

The state graphs of most stochastic models have many more arcs than states. This is especially the case in large systems that have many transitions, several of which may be enabled in each state. Further, even moderately complex systems have long state descriptors of tens or hundreds of bytes, making it more efficient to transmit hash keys instead of states wherever possible. By exploiting these two properties, we can dramatically reduce the amount of communication performed by our algorithm.

The algorithm described in the previous section (see Fig. 4.5) determines the identity of a non-local successor state s' by sending the full state descriptor s' to the remote node $h_0(s')$. Node $h_0(s')$ inserts state s' into its local hash table if it is not already present, and then returns the identity of state s'. This action is performed for every arc in the state graph that leads to a non-local successor state. In most instances, however, state s' will already be present in the hash table (since the number of arcs in the state graph is usually much greater than the number of states), so sending the full state descriptor was unnecessary. It suffices in such cases to send only the hash keys required to look up the state identity in the hash table.

Fig. 4.7 shows a communication-efficient algorithm which sends a minimal number of full state descriptors between nodes. As before, node i proceeds by popping a state off the local FIFO queue F_i and determining the set of successor states. The function child(s, n) is used to determine the nth successor state of a parent state s. Successor states for which $h_0(s) = i$ are dealt with locally in the usual fashion; other successor states are temporarily stored in a small state lookup table via a call to store-state(g, n, s') while their hash keys are sent to their owner processors via a call to send-keys(k, g, n, l, m). Here g is the identity of the parent state, n is the child number, s' is the child state and k, l, m are the partitioning, primary and secondary key values of s' respectively.

```
begin
    if h_0(s_0) = i do begin
        H_i = {[h_1(s_0), h_2(s_0)]}
        F_i.push(s_0)
        E_i = {s_0}
    end else
        H_i = E_i = ∅
    A_i = ∅
    while (shutdown signal not received) do begin
        if (F_i not empty) do begin
            s = F_i.pop()
            for n = 1 to |succ(s)| do begin
                s' = child(s, n)
                if h_0(s') = i do begin
                    if [h_1(s'), h_2(s')] ∉ H_i do begin
                        H_i = H_i ∪ {[h_1(s'), h_2(s')]}
                        F_i.push(s')
                        E_i = E_i ∪ {s'}
                    end
                    A_i = A_i ∪ {id(s) → id(s')}
                end else do begin
                    store-state(id(s), n, s')
                    send-keys(h_0(s'), id(s), n, h_1(s'), h_2(s'))
                end
            end
        end
        while (receive-keys(k, g, n, l, m)) do begin
            if [l, m] ∉ H_i do begin
                H_i = H_i ∪ {[l, m]}
                send-id(k, g, retrieve-id(l, m), n, false)
            end else
                send-id(k, g, retrieve-id(l, m), n, true)
        end
        while (receive-id(k, g, h, n, f)) do begin
            A_i = A_i ∪ {g → h}
            if (f = false)
                send-state(k, retrieve-state(g, n))
            delete-state(g, h)
        end
        while (receive-state(k, s')) do begin
            F_i.push(s')
            E_i = E_i ∪ {s'}
        end
    end
end
```

Figure 4.7: Communication-efficient state space generation algorithm for node i

Remote processors receive incoming hash keys via matching calls to receive-keys(k, g, n, l, m) where k is the sender node. If the hash keys $[l, m]$ are present in the hash table, the state is deemed to have been explored and its identity is retrieved from the hash table via a call to retrieve-id(l, m). This identity is returned to the parent node by calling send-id(k, g, h, n, f) where h is the identity of the child state and f is a flag set to true to indicate that the state was present in the hash table. If the hash keys are not present in the hash table, the state has not yet been encountered. Even so, the hash keys for the state are inserted into the hash table, and an identity is issued. This identity is returned to the parent node by calling send-id(k, g, h, n, f) where f is now set to false to indicate that the state was not present in the hash table. The parent processor needs to respond by sending this state so that it can be inserted into the remote processor's FIFO queue.

The parent processor receives returned identities via matching calls to receive-id(k, g, h, n, f) where k is the sender node. If the present flag f is not set, the original successor state descriptor s' stored in the state lookup table is retrieved using retrieve-state(g, h); the state s' is then sent to node k via a call to send-state(k, s'). Since the state descriptor s' is now no longer required by the parent processor, delete-state(g, h) removes it from the state lookup table.

Finally, incoming states are received via calls to receive-state(k, s') where k is the sender node. Incoming states are added to the local FIFO queue, and are written out to disk as part of the state space. Note how the activities of adding the state to the state hash table and adding it to the local FIFO queue have been separated. Also note that the algorithm is efficient in the sense that at most one copy of each full state descriptor is ever sent.

Fig. 4.8 summarises the main steps that the communication-efficient algorithm performs to identify each remote child s' of state s.

Given a state graph consisting of a arcs and n states, and assuming a uniform distribution of states across N processors, the communication cost (in bytes) of

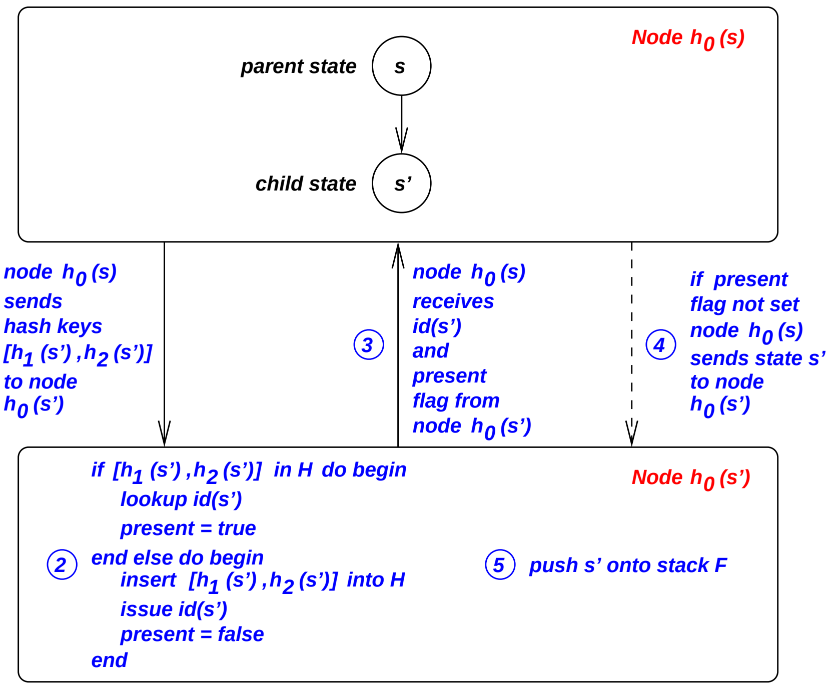


Figure 4.8: Steps required by the communication-efficient algorithm to identify child state s' of parent s.

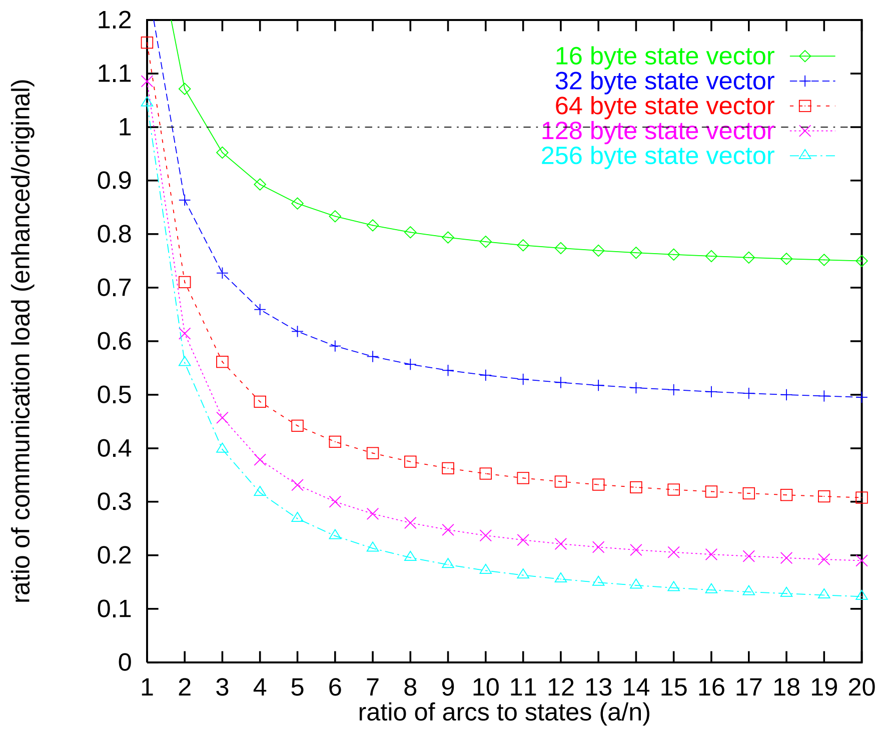


Figure 4.9: Ratio of the communication load of the enhanced algorithm to that of the original algorithm.

our enhanced algorithm is:

$$C(a, n, N) = \frac{(N-1)}{N}\left(a(C_k + C_i) + nC_s\right)$$

where C_k, C_i and C_s reflect the cost of sending a hash key, a state identity and a state descriptor respectively. Again, for efficiency reasons, real implementations (like ours) should buffer hash key, state and identity messages and send them in large blocks. Fig. 4.9 compares the ratio of the communication load of the enhanced algorithm to that of the original algorithm for various state graph densities (as given by the ratio of arcs to states a/n) and a range of state descriptor lengths. Using parameters taken from our implementation described in Chapter 6, we set $C_k = 12$ and $C_i = 8$. The graph shows that the enhanced algorithm reduces the communication load substantially for graphs with a ratio of arcs to states greater than 2 and that the reduction in communication load increases with increasing state descriptor size and increasing graph density. For graphs with a very low ratio of arcs to states and a short state descriptor, the communication load in fact increases. However, models with such low ratios are rare, and most of them correspond to simple queueing models that have simple product-form solutions.

4.6 Theoretical Performance Model

We now develop a model for predicting the run-time and speedup of the original and enhanced algorithms when implemented on a statically-routed wraparound mesh of N processors. We assume that there are a arcs to be generated and that there are a total of n unique states (nodes) in the state graph.

The model is based on the calculation of two key quantities: the *computation time* $T_W(a, n, N)$ required to generate arcs and search for states in the local hash table, and the *communication time* $T_C(a, n, N)$ required to send and receive non-local states. Predicted run-time $T_R(a, n, N)$ is then simply given by

$$T_R(a, n, N) = T_W(a, n, N) + T_C(a, n, N)$$

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 balancing 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. 4.5 (for the original algorithm) and Fig. 4.7 (for the enhanced algorithm).

We assume 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 $T_W(a, n, N)$ is thus estimated by:

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

The number of non-local arcs m generated per processor, assuming that new destination states 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. For the original algorithm,

$$L = C_i + C_s$$

where C_i and C_s are the costs of sending a state identity and a state descriptor respectively. For the enhanced communication-efficient algorithm,

$$L = C_k + C_i + \frac{nC_s}{a}$$

where C_k is the cost of sending a set of hash keys.

To prevent the communication network from being overwhelmed by thousands of short messages, state, identity and hash key messages are buffered and sent in blocks between processors. The overhead associated with buffer management for each block (i.e. the time spent packing and unpacking messages) 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. The total number of buffers that need to be sent is mL/B.

In Appendix B.4.1 we show that the time taken to communicate a message between two nodes in a 2D wraparound mesh with static wormhole routing is given by:

$$t_c(l) = t_s(l) + t_h(x, y, t_1, t_2) + t_b(l, \lambda)$$

where:

- l is the message length in bytes.
- x and y are the dimensions of the mesh.
- t_1 and t_2 are the per-hop flit latencies for transfers in the same direction (i.e. X→X or Y→Y) and different directions (i.e. X→Y) respectively.
- λ is the maximum throughput of the underlying network.
- $t_s(l)$ is the startup time for a message of length l bytes (usually proportional to l).
- $t_h(x, y, t_1, t_2)$ is the average hop time required for a header to travel between sender and receiver. For a square mesh of N processors with $x = y = \sqrt{N}$ and a per-hop flit latency of $t_1 = t_2 = f$ seconds,

  $$t_h(N, f) \approx \frac{\sqrt{N}}{2} f.$$

Appendix B.4.1 gives an exact formula for the general case of an $x \times y$ mesh with $t_1 \neq t_2$.

- $t_b(l, \lambda) = \lambda l$ is the transmission time that a message of l bytes requires to traverse the link between sender and receiver.

Using this model, the total time T_C spent by each processor on communication overhead is, on average:

$$T_C(a, n, N) = m\left(\frac{L(s + t_c(B))}{B}\right) = \frac{a(N-1)}{N^2}\left(\frac{(s + t_c(B))L}{B}\right)$$

For a square mesh with a per-hop flit latency of f seconds, we have

$$T_C(a, n, N) = \frac{a(N-1)}{N^2}\left((s + t_s(B) + (\sqrt{N}/2)f + \lambda B)\frac{L}{B}\right)$$

The speedup of the algorithm executing on this architecture can now be calculated as:

$$\begin{aligned} S(a, n, N) &= T_W(a, n, 1)/T_R(a, n, N) \\ &= T_W(a, n, 1)/\left(T_W(a, n, N) + T_C(a, n, N)\right) \end{aligned}$$

and its efficiency is given by $E(a, n, N) = S(a, n, N)/N$;

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:

$$C(a, n, N) = N\left(T_W(a, n, N) + T_C(a, n, N)\right)$$

which cannot be proportional to $T_W(a, n, 1)$ for large N, on account of the $\sqrt{N}/2$ term in $T_C(a, n, N)$. 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 $(\sqrt{N}/2)f$ grows slowly and is typically negligible in comparison with λB for moderate N, the algorithm's efficiency is maintained well for machines with up to a few hundred processors.

4.7 Results

We have implemented the state generation algorithms of Fig. 4.5 and Fig. 4.7 on a Fujitsu AP3000 distributed memory parallel computer. Our implementation is written in C++ with support for two popular parallel programming interfaces, viz. the Message Passing Interface (MPI) [GLS94] and the Parallel Virtual Machine (PVM) interface [GBD+94]. The generator uses hash tables with $r = 750\,019$ rows per processor and $b = 40$ bit secondary keys. Models are specified using the interface language described in Chapter 6 which allows for the high-level specification of a wide variety of stochastic models. The high-level specification is then translated into a C++ class which is compiled and linked with a library implementing the core state generator. The state space and state graph are written to disk in compressed format as the algorithm proceeds.

The results were collected using up to 16 processors on the AP3000. Each processor has a 300MHz UltraSPARC processor, 256MB RAM and a 4GB local disk. The nodes run the Solaris operating system and support MPI. They are connected by a high-speed wormhole-routed network with a peak throughput of 65MB/s.

Interested readers can find further details of the implementation in Chapter 6, while more details of the machine and its configuration are given in Appendix B.

4.7.1 The FMS Model

The first example we consider is a 22-place GSPN model of a flexible manufacturing system. This model, hereafter referred to as the FMS model, was initially described in [CT93] and was subsequently used in [CGN98] to demonstrate distributed exhaustive state space generation. A full description of this model is presented in Appendix A.2, although a complete understanding of the model is not required here. It suffices to note that the model has a parameter k (corresponding to the initial number of parts in the system), and that as k increases,

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

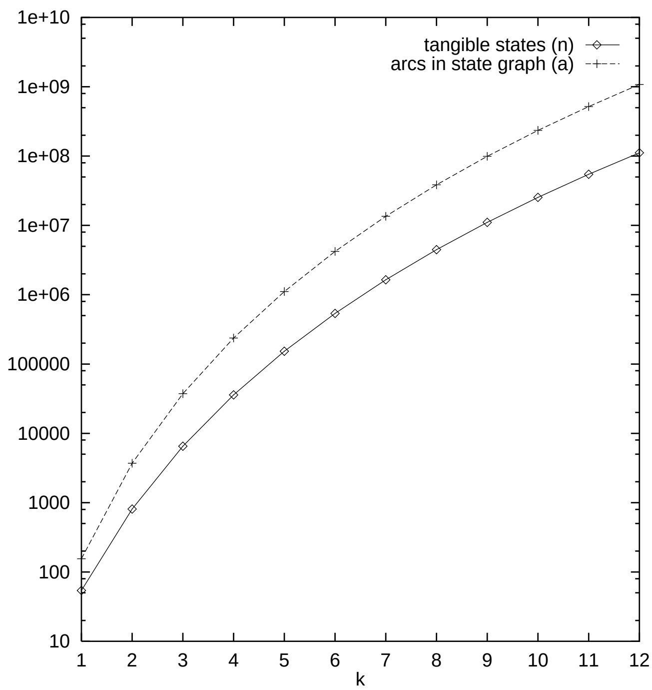


Figure 4.10: The number of tangible states (n) and the number of arcs (a) in the state graph of the FMS model for various values of k.

so do the number of states n and the number of arcs a in the state graph (see Fig. 4.10).

Run times and speedups

The graph on the left in Fig. 4.11 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, 12 and 16 processors on the AP3000. Each observed run-time value is calculated as the mean of four runs. The $k = 8$ state space (4 459 455 states) can be generated on a single processor in about 13 minutes 30 seconds; 16 processors require just 71 seconds. The $k = 9$ state space (11 058 190 states) can be generated on a single processor in 36 minutes; 16 processors require just 182 seconds.

The graph on the right in Fig. 4.11 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 16 processors we observe a speedup of 11.85, giving an efficiency of 74%. 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, as predicted by the analysis of Section 4.6.

Fig. 4.12 shows the corresponding run times and speedups for the communication-efficient algorithm. For $k = 9$ using 16 processors, the run time is 160 seconds, a 12% improvement over the original algorithm. The corresponding speedup on 16 processors is 13.47, giving an efficiency of 84% (vs. 74% for the original algorithm).

Memory utilization is low – a single processor generating the $k = 8$ state space uses a total of 82MB RAM (18.8 bytes per state) while the $k = 9$ state space requires 141MB RAM (13.1 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

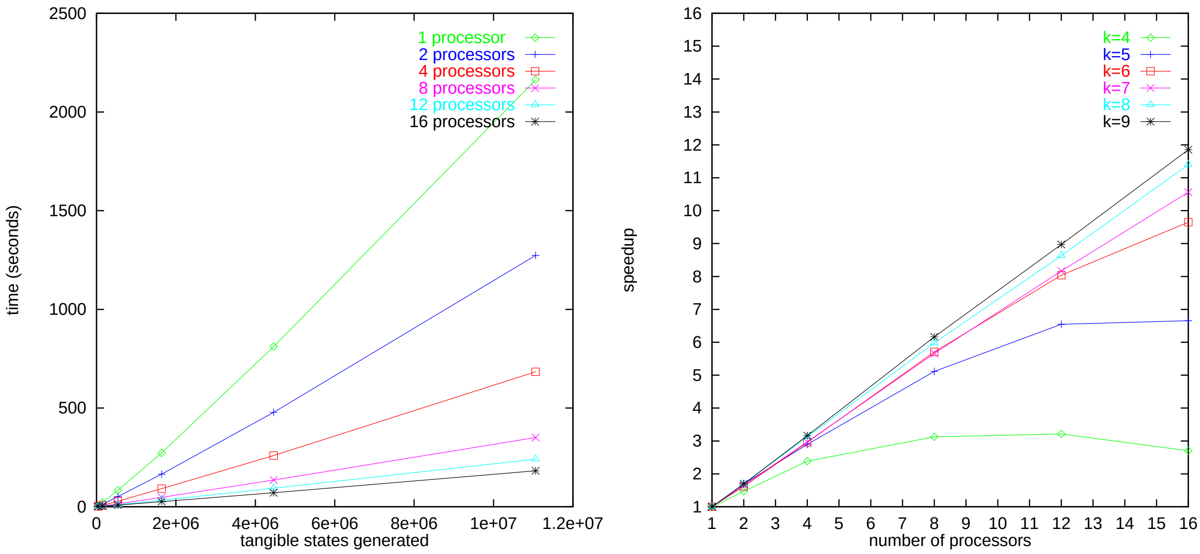


Figure 4.11: Real time taken to generate FMS state spaces up to $k = 9$ using the original algorithm on 1, 2, 4, 8, 12 and 16 processors (left), and the resulting speedups for $k = 4, 5, 6, 7, 8$ and 9 (right)

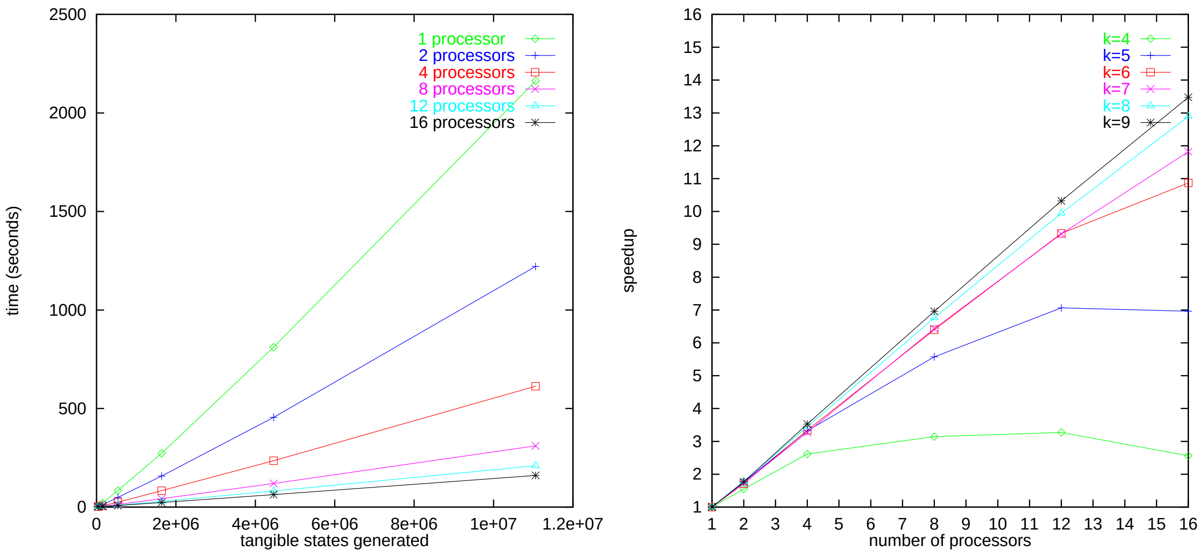


Figure 4.12: Real time taken to generate FMS state spaces up to $k = 9$ using the communication-efficient algorithm on 1, 2, 4, 8, 12 and 16 processors (left), and the resulting speedups for $k = 4, 5, 6, 7, 8$ and 9 (right)

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 would be even more marked with a complex model that has a longer state descriptor, since the memory consumption of our technique is independent of the number of elements in the state descriptor.

Larger state graphs

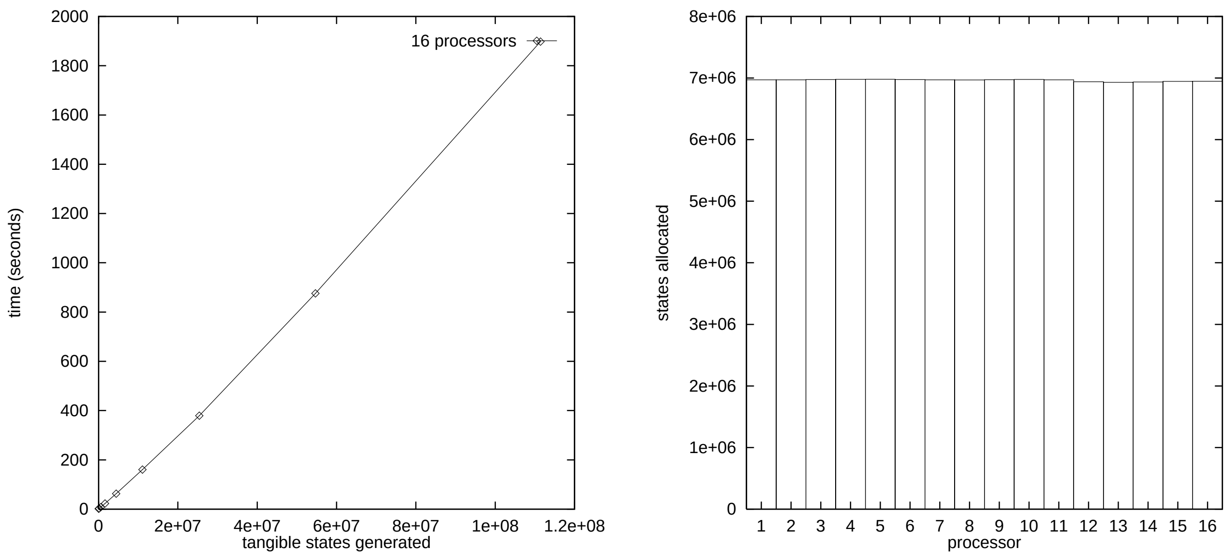


Figure 4.13: Real time taken to generate state spaces up to $k = 12$ using 16 processors (left) and distribution of states across processors for $k = 12$ (right).

Moving beyond the maximum state space size that can be generated on a single processor, the graph on the left in Fig. 4.13 shows the real time required to generate larger state spaces using the communication-efficient algorithm running on 16 processors. For the largest case ($k = 12$), under 32 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. 4.13 shows the distribution of the number of states generated by each processor for the case $k = 12$.

In comparison to the results reported above, 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 [CGN98]. They were unable to explore state spaces for larger values of k (c.f. Fig 4.10).

To enhance our confidence in our results for the case $k = 12$, we use Eq. 4.9 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:

$$q \approx \frac{n^2}{Nr2^{b+1}} = \frac{10^{16}}{16 * 750\,019 * 2^{40}} = .000379$$

i.e. the omission probability is less than 0.05%. 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. It is important to stress that the omission probability applies to the entire state graph and does *not* denote the average proportion of states that will be omitted or misidentified. That is, there is a 99.95% chance that the state graph has been generated in its entirety without omitting or misidentifying any of the 10^8 states whatsoever.

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.

Validation of the performance model

The accuracy of the performance model presented in Section 4.6 has been assessed by comparing observed runtimes and speedups with model predictions for the FMS model. The model parameters used are given in Fig. 4.14. Note that the model parameters for both the original and enhanced algorithms are exactly the same, except for L, the amount of traffic generated by the processing of a non-local arc.

Fig. 4.15 and Fig. 4.17 show predicted and observed run times over various state space sizes for the original and enhanced algorithms respectively. Predicted run-

	Parameter description	Value
N	number of nodes	(variable)
a	number of arcs in state graph	(variable)
n	number of states in state space	(variable)
L	comms induced by one non-local arc – original algorithm	56 bytes
	comms induced by one non-local arc – enhanced algorithm	$20 + 48n/a$ bytes
B	message buffer size	8192 bytes
F	flit size	32 bits
$x \times y$	mesh dimensions	8×4
t_1	per-hop flit latency – same direction (X→X or Y→Y)	120ns
t_2	per-hop flit latency – change direction (X→Y)	170ns
λ	underlying network throughput	65.6MB/s
c	cost of generating one arc	20.5 μs
d	cost of scanning one hash table entry	180 ns
s	cost of buffer management (per buffer)	1050 μs
r	number of hash table rows per node	750 019 rows

Figure 4.14: Parameters used in the FMS performance model

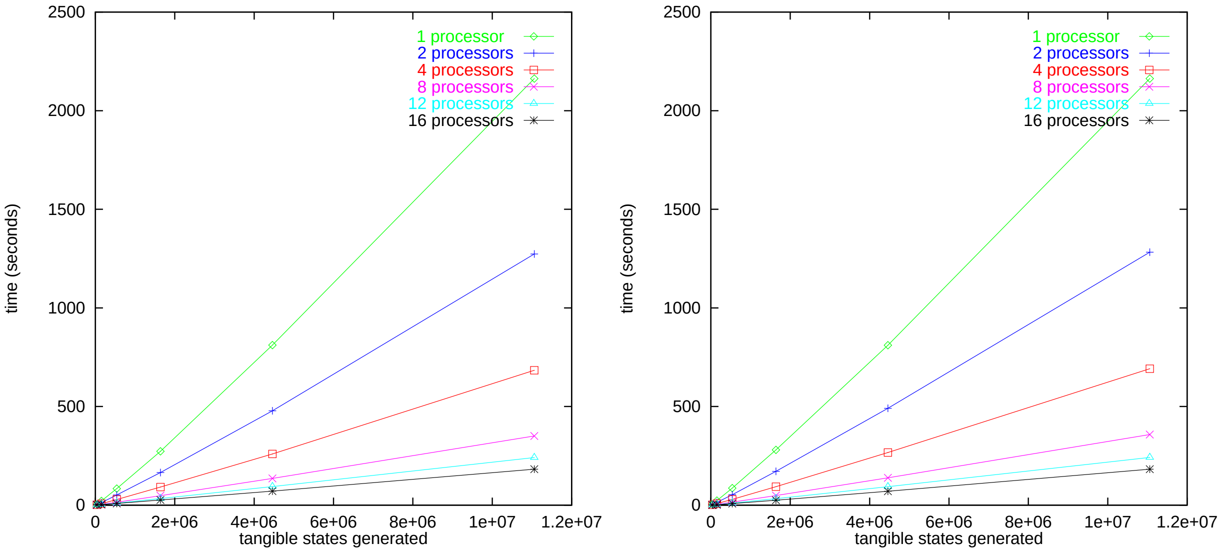


Figure 4.15: Observed (left) and predicted (right) real time taken to generate FMS state spaces up to $k = 9$ using the original algorithm on 1, 2, 4, 8, 12 and 16 processors

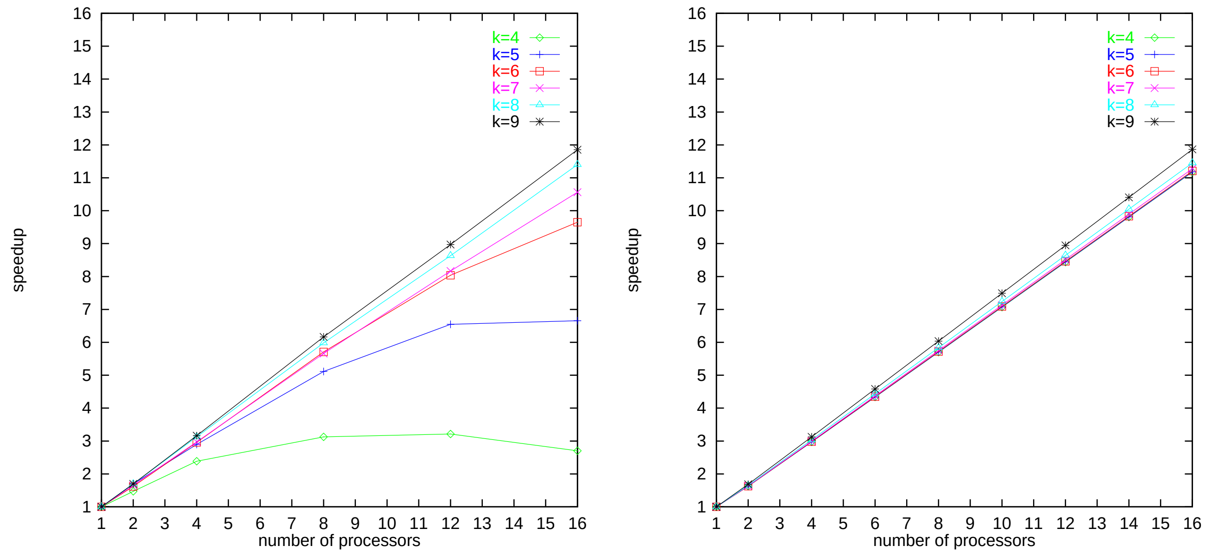


Figure 4.16: Observed (left) and predicted (right) FMS speedups for the original algorithm with $k = 5, 6, 7, 8$ and 9

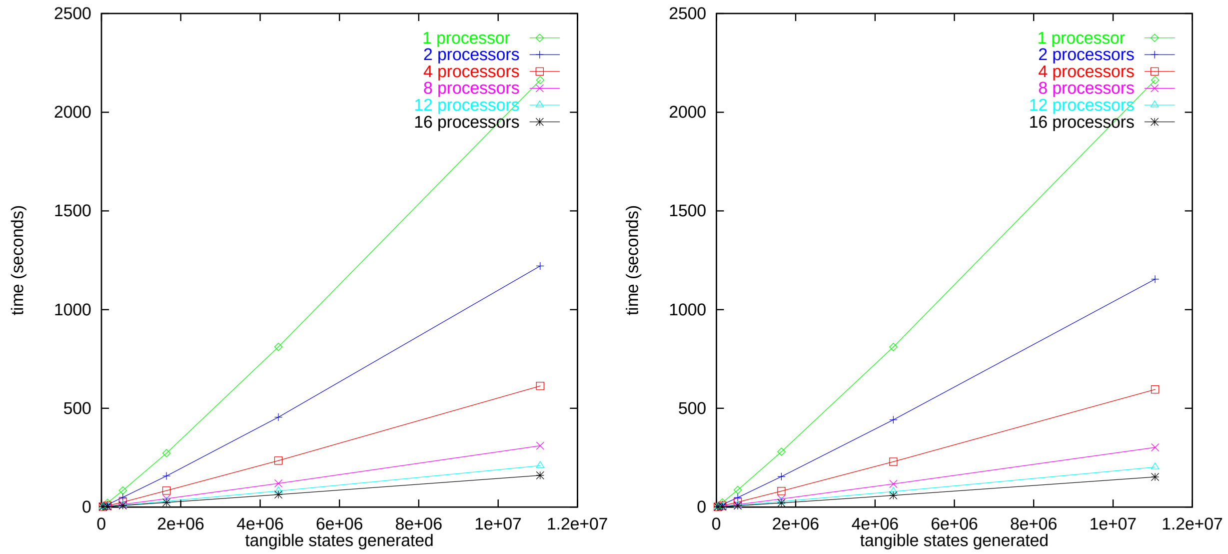


Figure 4.17: Observed (left) and predicted (right) real time taken to generate FMS state spaces up to $k = 9$ using the communication-efficient algorithm on 1, 2, 4, 8, 12 and 16 processors

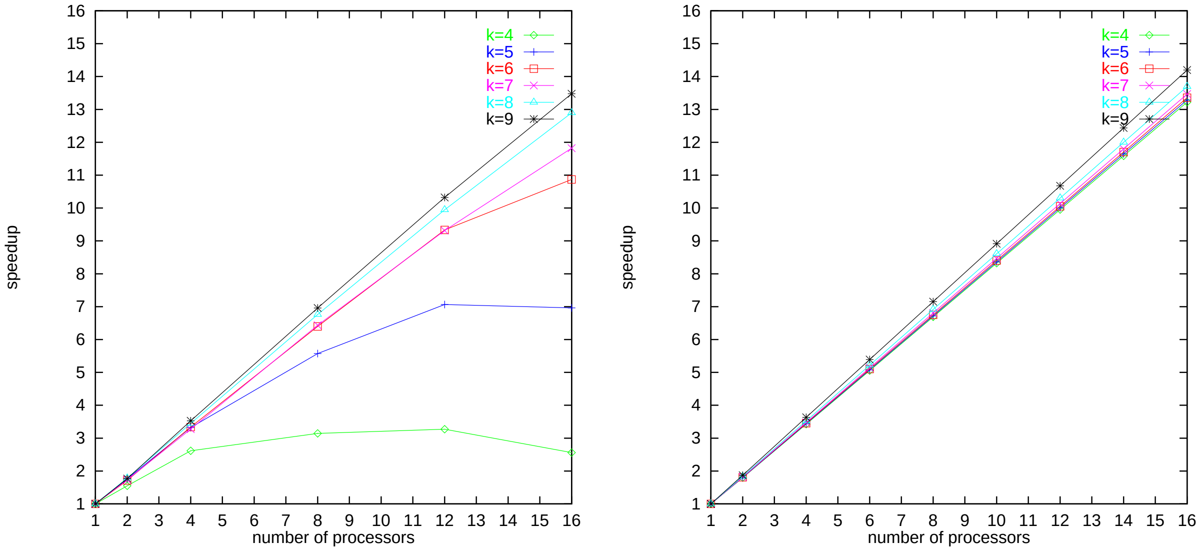


Figure 4.18: Observed (left) and predicted (right) FMS speedups for the communication-efficient algorithm for $k = 5, 6, 7, 8$ and 9

times for the single processor case, which does not involve any communication, are typically within 3% of the observed values, suggesting that our model for $T_W(a, n, N)$ is accurate. For the larger state spaces ($k = 7, 8, 9$) predicted run-times for multiple processor runs involving communication are typically within 5% of the observed values, suggesting that our model for $T_C(a, n, N)$ is also accurate for large state spaces. However, for smaller state spaces ($k = 4, 5, 6$), there is a tendency for the model to predict significantly faster distributed run times than are actually observed. This trend is clearly evident in Fig. 4.16 and Fig. 4.18 which show predicted and observed speedups for the original and enhanced algorithms respectively. This is not surprising since our model does not take into account the start-up and termination phases which require a significant proportion of run-time for small state spaces.

4.7.2 The Courier Protocol Model

The second example we consider is a 45-place GSPN model of software used in the Courier telecommunications protocol [WL91]. Full details of the model are given in Appendix A.3. As was the case for the FMS model, there is a scaling parameter k (corresponding to the sliding window size) which we will vary to produce state graphs of different sizes (see Fig. 4.19).

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

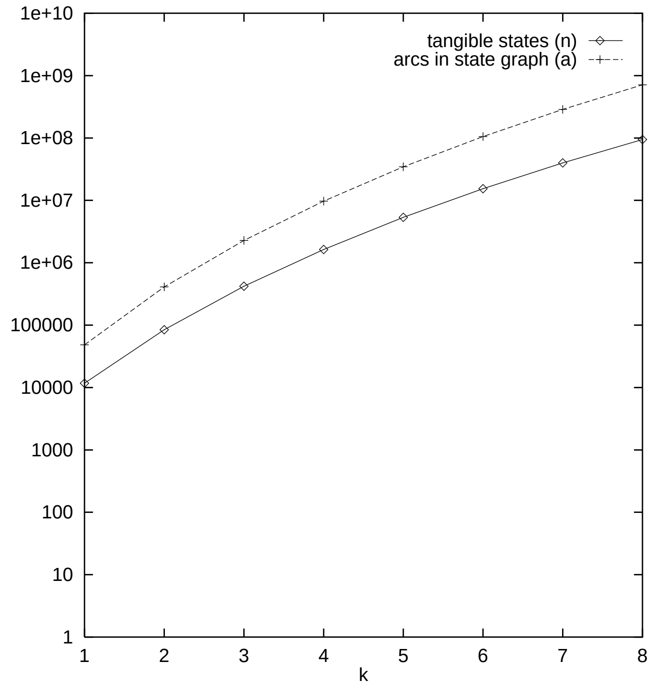


Figure 4.19: The number of tangible states (n) and the number of arcs (a) in the state graph of the Courier model for various values of k.

Run times and speedups

The graph on the left of Fig. 4.20 shows the distributed run-time taken to explore Courier state spaces of various sizes (up to $k = 6$) using 1, 2, 4, 8, 12 and 16 processors on the AP3000. As for the FMS model, each observed value is calculated as the mean of four runs. The $k = 5$ state space (5 358 600 states) can be generated on a single processor in 16 minutes 20 seconds; 16 processors require only 89 seconds. The $k = 6$ state space (15 410 250 states) can be generated on a single processor in 51 minutes 45 seconds; 16 processors require just 267 seconds.

The corresponding speedups for the cases $k = 1, 2, 3, 4, 5, 6$ are shown in the graph on the right of Fig. 4.20. For $k = 6$ using 16 processors, we observe a speedup of 11.65, giving an efficiency of 73%.

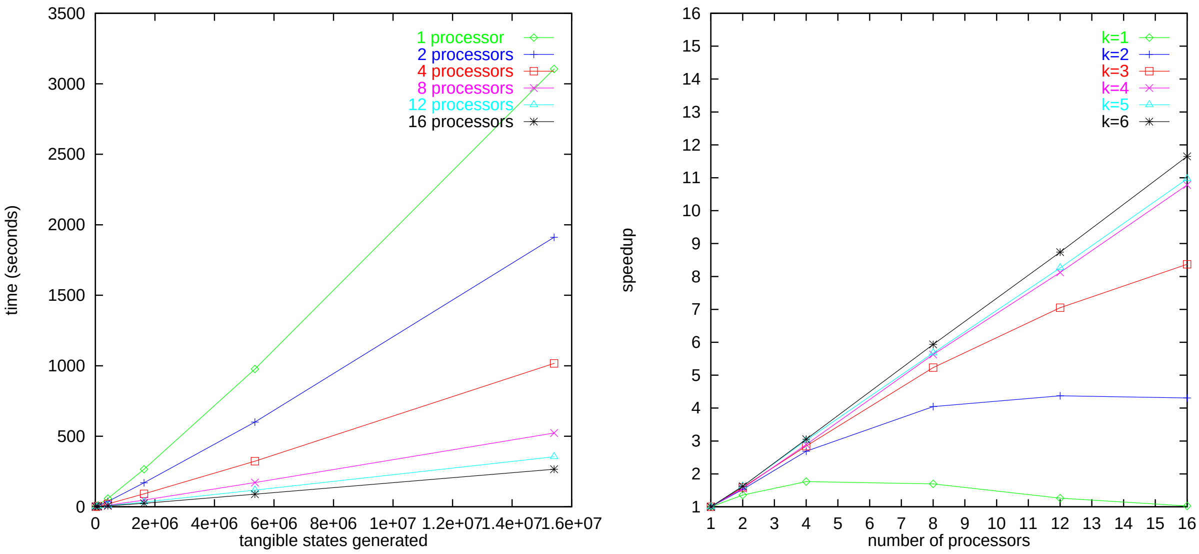


Figure 4.20: Real time taken to generate Courier state spaces up to $k = 6$ using the original algorithm on 1, 2, 4, 8, 12 and 16 processors (left), and the resulting speedups for $k = 1, 2, 3, 4, 5$ and 6 (right).

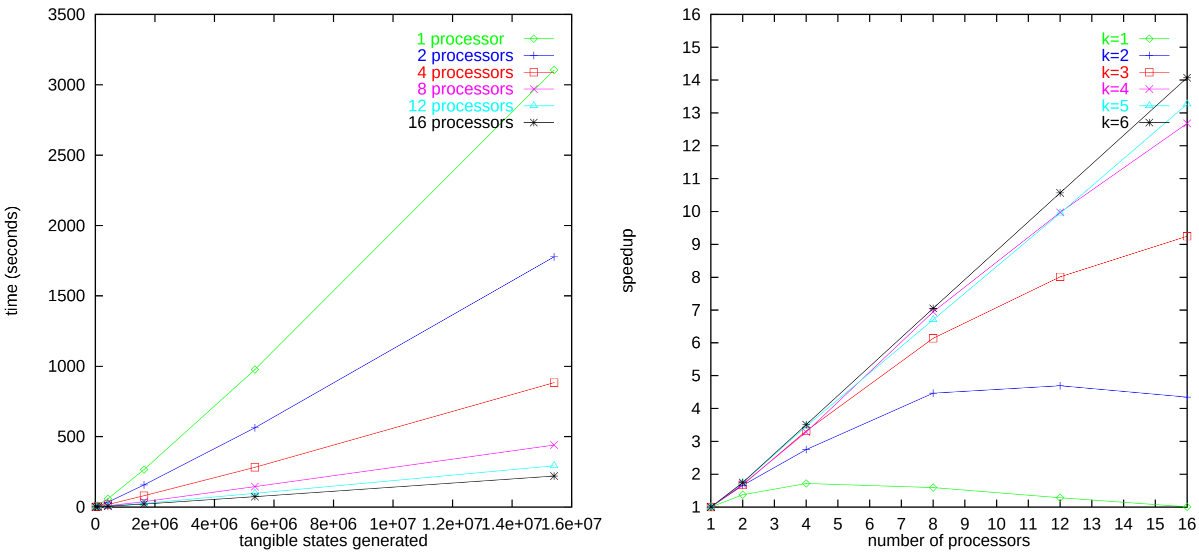
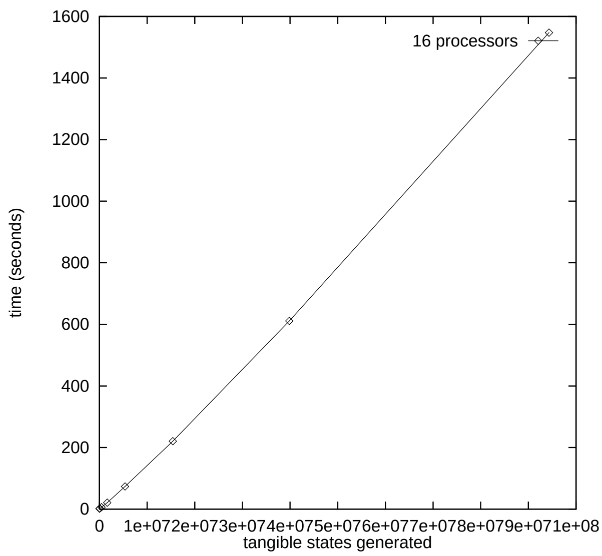


Figure 4.21: Real time taken to generate Courier state spaces up to $k = 6$ using the communication-efficient algorithm on 1, 2, 4, 8, 12 and 16 processors (left), and the resulting speedups for $k = 1, 2, 3, 4, 5$ and 6 (right).

Fig 4.21 shows the corresponding run times and speedups for the communication-efficient algorithm. For $k = 6$ using 16 processors, the run time is 221 seconds, a 17% improvement over the original algorithm. The corresponding speedup on 16 processors is 14.07, giving an efficiency of 88% (vs. 73% for the original algorithm). The enhanced algorithm delivers a better performance improvement here than it did in the case of the FMS model because the Courier model has a state descriptor that is about double the size of the FMS model (45 integers for the Courier model vs. 22 integers for the FMS model). Consequently the enhanced algorithm eliminates a greater proportion of the communication overhead (c.f. Fig. 4.9).

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

Larger state graphs



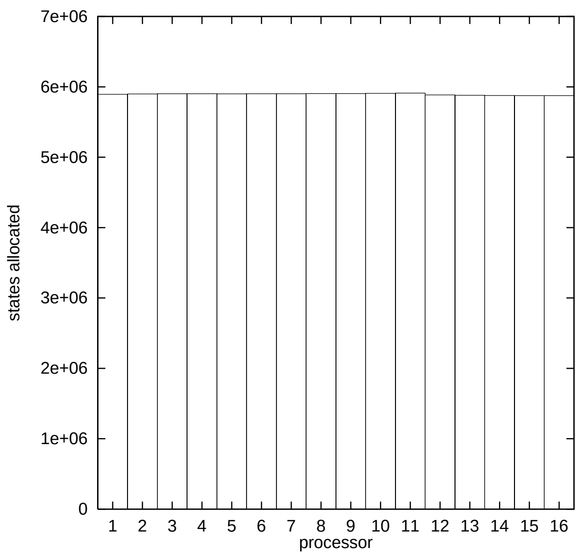


Figure 4.22: Real time taken to generate state spaces up to $k = 8$ using 16 processors (left) and distribution of states across processors for $k = 8$ (right)

Moving beyond the maximum state space size that can be generated on a single processor, the graph on the left of Fig. 4.22 shows the real time required to generate larger state spaces using the communication-efficient algorithm running on 16 processors. For the largest case ($k = 8$), under 26 minutes are required to generate a state space with 94 322 250 states and a state graph with 710 223 930 arcs. The graph on the right of Fig. 4.22 shows the corresponding assignment of states to processors for the case $k = 6$.

As for the FMS model, the omission probability is very low (less than 0.05%), and can be arbitrarily improved by repeatedly regenerating the state space with independent sets of hash functions.

Validation of the performance model

	Parameter description	Value
N	number of nodes	(variable)
a	number of arcs in state graph	(variable)
n	number of states in state space	(variable)
L	comms induced by one non-local arc – original algorithm	102 bytes
	comms induced by one non-local arc – enhanced algorithm	$20 + 94n/a$ bytes
B	message buffer size	8192 bytes
F	flit size	32 bits
$x \times y$	mesh dimensions	8×4
t_1	per-hop flit latency – same direction (X→X or Y→Y)	120ns
t_2	per-hop flit latency – change direction (X→Y)	170ns
λ	underlying network throughput	65.6MB/s
c	cost of generating one arc	27.5 μs
d	cost of scanning one hash table entry	180 ns
s	cost of buffer management (per buffer)	1050 μs
r	number of hash table rows per node	750 019 rows

Figure 4.23: Parameters used in the Courier performance model

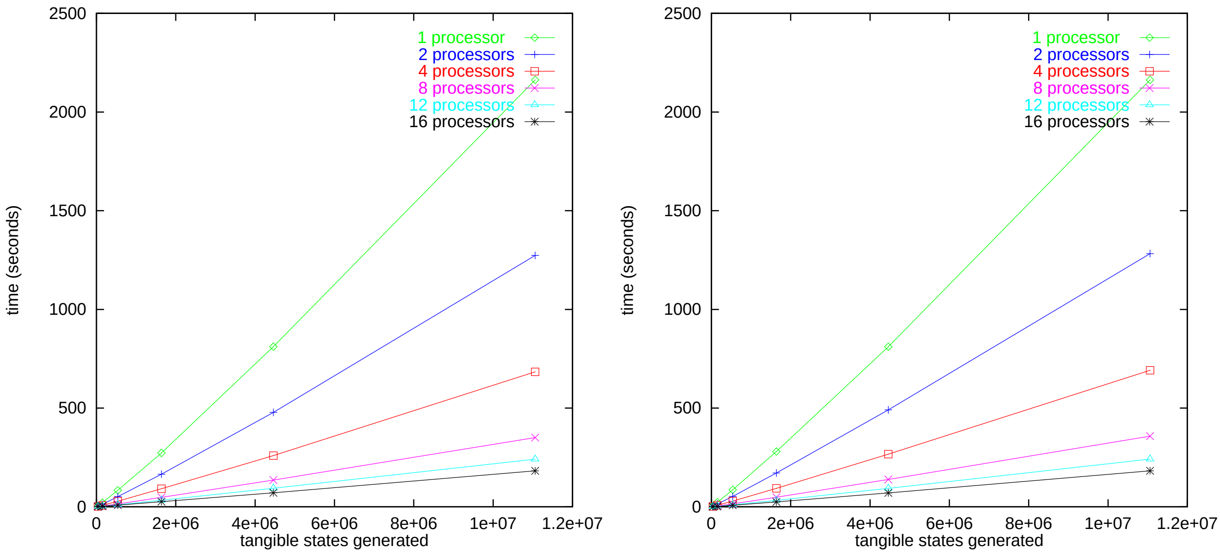


Figure 4.24: Observed (left) and predicted (right) real time taken to generate Courier state spaces up to $k = 9$ using the original algorithm on 1, 2, 4, 8, 12 and 16 processors

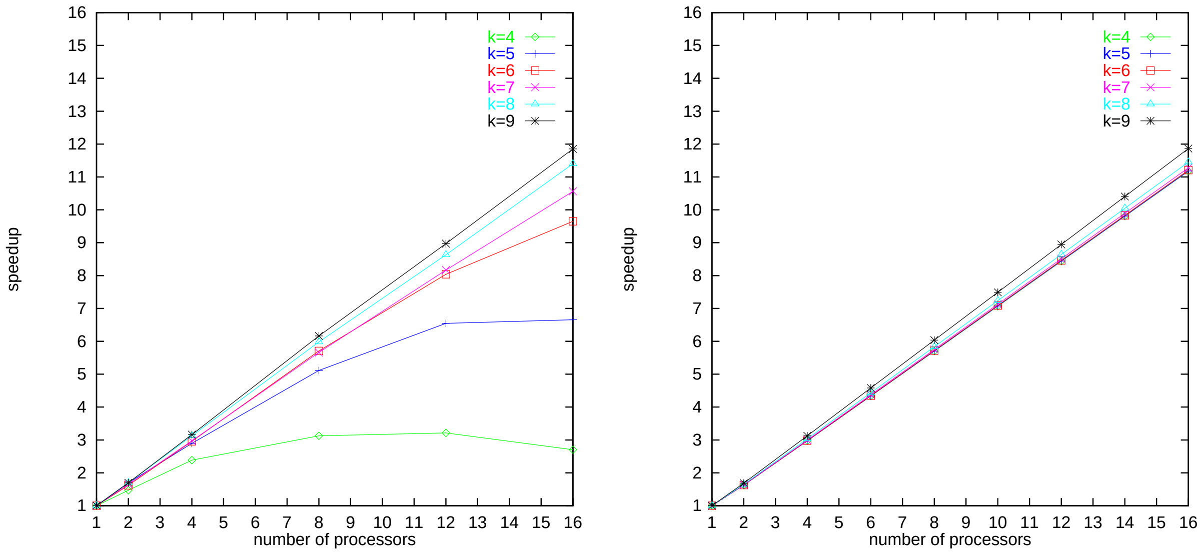


Figure 4.25: Observed (left) and predicted (right) Courier speedups for the original algorithm with $k = 5, 6, 7, 8$ and 9

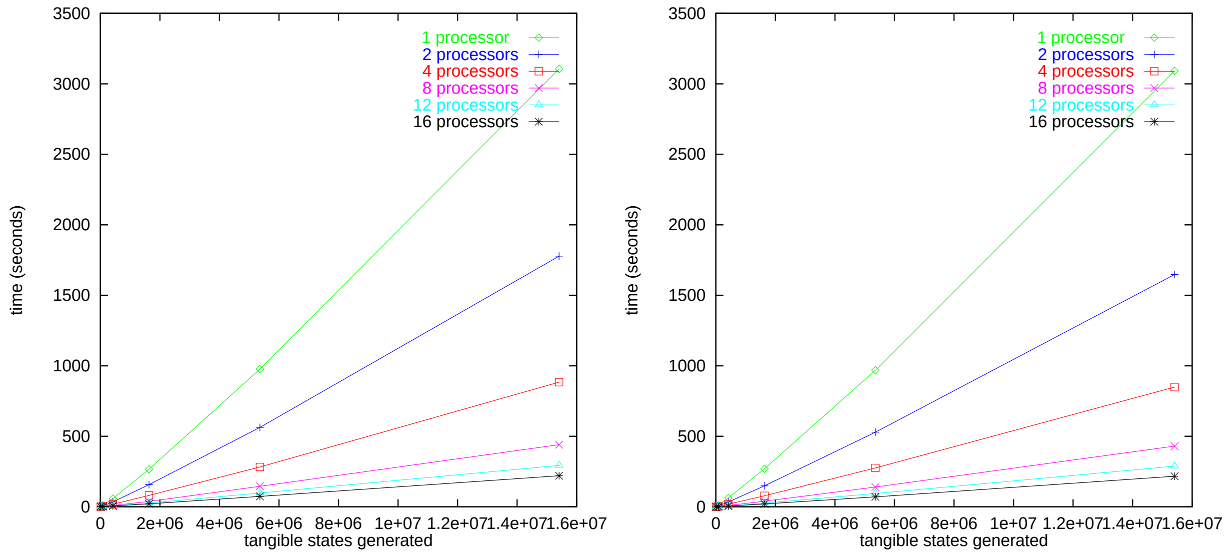


Figure 4.26: Observed (left) and predicted (right) real time taken to generate Courier state spaces up to $k = 9$ using the communication-efficient algorithm on 1, 2, 4, 8, 12 and 16 processors

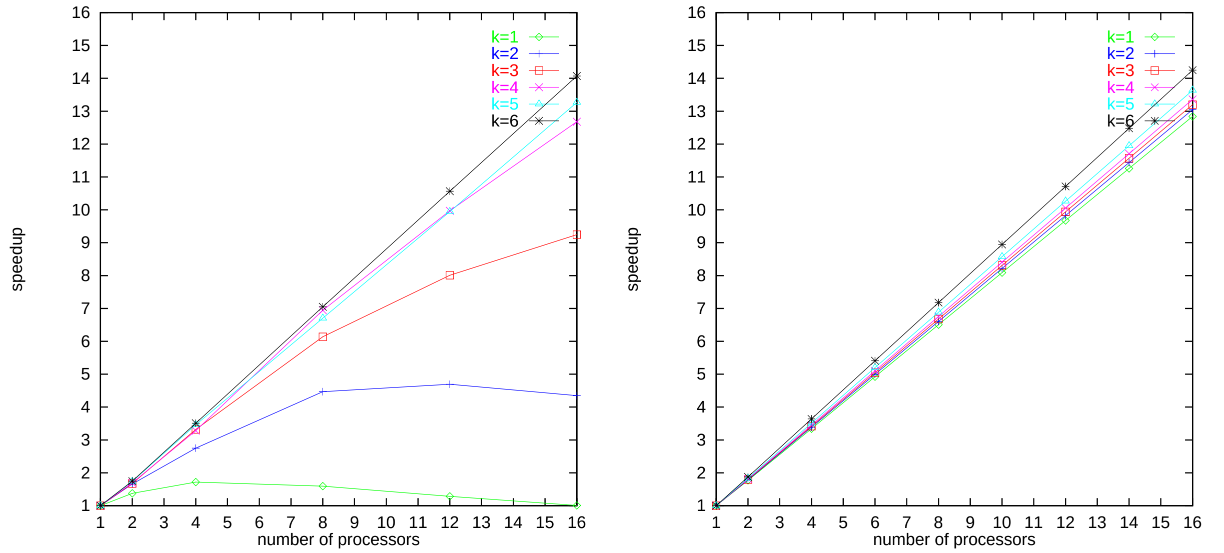


Figure 4.27: Observed (left) and predicted (right) Courier speedups for the communication-efficient algorithm for $k = 5, 6, 7, 8$ and 9

This section compares the runtimes and speedups predicted by the theoretical performance model of Section 4.6 with the observed results for the Courier model. The model parameters used are given in Fig. 4.23. The parameters used to predict the Courier model differ from those used for the FMS model in only two respects. Firstly, the cost of generating an arc c is higher, since the Courier model has a greater number of transitions and a higher proportion of vanishing states that need to be eliminated. Secondly, the amount of traffic generated by the processing of a non-local arc L differs since the two models have different state vector sizes. As for the FMS model, the set of parameters used to make predictions for both the original and enhanced algorithms is the same, with the exception of L.

Fig. 4.24 and Fig. 4.26 show predicted and observed run times over various state space sizes for the original and enhanced algorithms respectively. For the larger state space sizes ($k = 4, 5, 6$) agreement for both the single and multiple processor run times is excellent (within 5%). For the smaller state spaces ($k = 1, 2, 3$) there is again a tendency for the model to predict faster distributed run times than are observed, because the model does not take into account start-up and termination overheads. Fig. 4.25 and Fig. 4.27 show the corresponding observed and predicted speedups. The predicted speedup is particularly accurate for large models, for both the original and enhanced algorithms. Predicted speedups for smaller models are too high because of the ideal model assumptions we have already mentioned.

4.8 Choosing Good Hash Functions

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

- the **partitioning hash function** $h_0(s) \to (0, 1, \ldots, N-1)$, which assigns state s to a processor.
- the **primary hash function** $h_1(s) \to (0, 1, \ldots, r-1)$ 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.

Note that, while it is theoretically impossible to define a hash function that creates random output data from nonrandom input data, in practice it is often possible to produce a good imitation of random output data [Knu98]. In fact, the Universal$_2$ class of hash functions developed by Carter and Wegman [CW79] was designed exactly for this purpose. The idea is to hash an l-word key

$$K = x_1 x_2 x_3 \ldots x_l$$

onto the value

$$h(K) = (h_1(x_1) + h_2(x_2) + \ldots + h_l(x_l)) \mod M$$

where each h_i is an independent hash function. Carter and Wegman prove that this strategy minimizes collisions *regardless of the input data.* Unfortunately, calculating l independent hash functions is too expensive for our purposes. Consequently, we need to develop hash functions which are based on similar principles as the Universal$_2$ class of functions, but which are cheaper to compute.

Before we consider each of the partitioning, primary and secondary hash functions individually, consider the two general hash functions f_1 and f_2 shown in Fig. 4.28. 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.

```
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
```

Figure 4.28: Two general hash functions for mapping states onto 32 bit unsigned integers.

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 an offset of $(i \times \mathit{shift})$ mod 32 bits. The result is a simple, rapid function that provides a reasonable distribution of hash values.

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 $i \times \mathit{shift}_1$ mod 32, but also the sum $\sum_{j<i} s_i$ rotated left by an offset of $i \times \mathit{shift}_2$ mod 32. This technique makes the hash function resistant to any symmetries and invariants that may be present in the model. This function involves more computation than f_1 but provides an excellent distribution of hash values.

We now make use of functions f_1 and f_2 to derive suitable choices for $h_0(s)$, $h_1(s)$ and $h_2(s)$. The results presented in Section 4.7 made use of partitioning and primary functions based on f_1 and a 40-bit secondary hash function based on f_2.

4.8.1 Partitioning Hash Function

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 $>> N$.

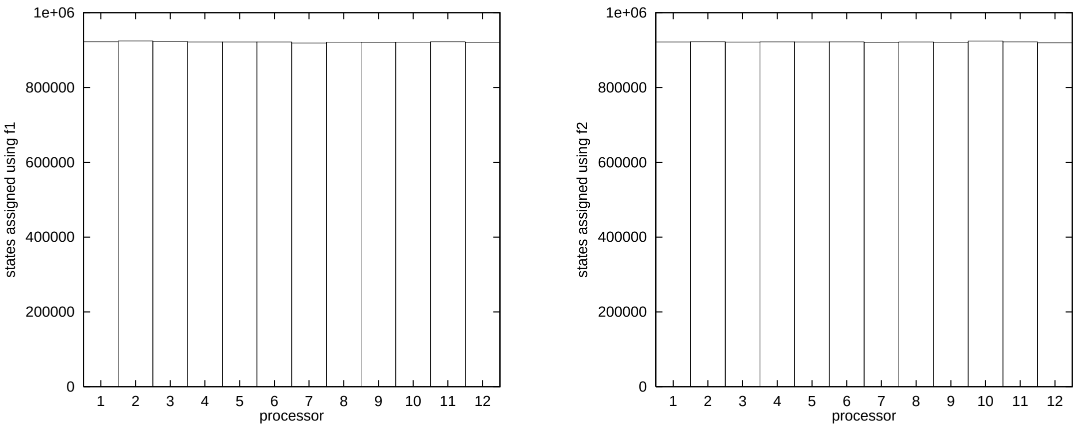


Figure 4.29: State distributions for the FMS model with $k = 9$ and $N = 12$ using $h_0(s) = f_1(s, 3) \bmod 5\,003 \bmod N$ (left) and $h_0(s) = f_2(s, 3, 5) \bmod 5\,003 \bmod N$ (right).

The graphs in Fig. 4.29 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 4.3 compares the performance of three partitioning hash functions over a wider range of k and N values. The functions considered are an ideal random hashing function, a function based on f_1 ($h_0(s) = f_1(s, 3) \bmod 5\,003 \bmod N$) and a function based on f_2 ($h_0(s) = f_2(s, 3, 5) \bmod 5\,003 \bmod N$). The performance is expressed in terms of σ_N, the standard deviation of the number of states assigned to each processor. We assume that the ideal random hash function

		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 4.3: Values of σ_N, the standard deviation of the number of states allocated to each processor, for the FMS model using three partitioning functions.

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 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.

Ciardo *et al* investigate the problem of choosing a good partitioning hash function that provides some measure of *locality* as well as good load balance [CGN98]. Locality means that most of a state's successors are assigned to the same processor as the parent state, resulting in a lower communication load. The communication load can be measured in terms of the number of *cross-arcs* between partitions, i.e. the number of states sent from one processor to another.

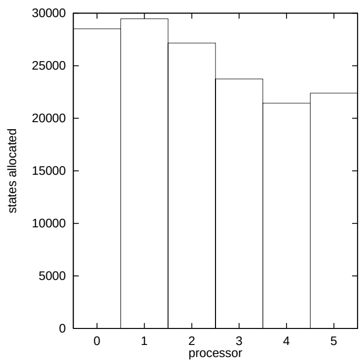
One way to achieve locality is to use a partitioning hash function based on a small subset of the state descriptor called the *control set*. Many of the transitions do not alter the control set and therefore do not create cross-arcs. Fig. 4.30(a) shows the communication patterns and state distributions that result from a partitioning hash function based on a control set.

Unfortunately, there are several problems with this approach. Firstly, the use of a control set often leads to an unsatisfactory load balance and uneven communication patterns that tend to deteriorate even further as the number of processors is

	0	1	2	3	4	5
0	104265	31123	10844	9345	9376	38528
1	47925	107937	30878	11797	10062	9588
2	7640	47830	97875	28272	11235	9027
3	6729	6103	42542	83385	25288	9698
4	7750	5694	5541	36516	73881	24239
5	27516	7962	6411	6494	33967	78219

$h_0(s) = (M(P1) + M(P2) \cdot 1013 + M(P3) \cdot 1013^2) \bmod N$

50.9% of arcs are cross-arcs

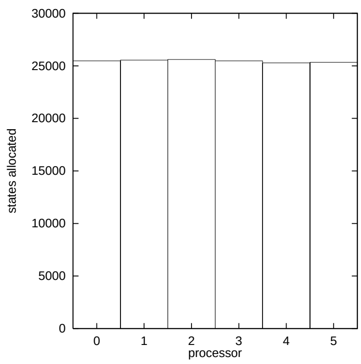


(a) Number of arcs from processor i to processor j (left) and distribution of states across processors (right) for a hash function based on a control set [CGN98].

	0	1	2	3	4	5
0	31639	28725	31642	28966	39539	24575
1	24938	31873	29438	31523	28608	39637
2	39851	24920	31643	29207	31591	29080
3	29120	39600	24948	31660	28884	31208
4	31136	28971	39455	24502	31144	28843
5	28846	31447	29093	39567	24238	31425

$h_0(s) = f_1(s, 3) \bmod 5003 \bmod N$

83.0% of arcs are cross-arcs



(b) Number of arcs from processor i to processor j (left) and distribution of states across processors (right) for a hash function based on f_1.

Figure 4.30: Communication patterns and state distributions for the FMS model ($k = 5, N = 6$) when using (a) a hash function based on a control set and (b) a hash function based on f_1.

increased. Secondly, there is at present no way to automatically select a suitable control set, so we have to rely on the user's intuition to provide one. Finally, in the context of our parallel dynamic hash compaction method, the use of a control set may weaken the reliability of our algorithm since many states with similar state descriptor elements will be assigned to the same processor. Quantifying this effect precisely is difficult since it will depend on the model and the form of the primary and secondary hash functions used.

Because of these problems, our partitioning hash function does not attempt to exploit locality. Instead we use a general hash function which can be automatically generated and which leads to a very good load balance and even communication patterns without affecting the reliability of our algorithm. Fig. 4.30(b) shows the balanced communication patterns and state distributions of our partitioning hash function.

Of course a user may wish to substitute a user-defined partitioning hash function if there is some application-specific feature which enables the exploitation of locality while preserving both good load balance and the reliability of the algorithm.

4.8.2 Primary Hash Function

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.

Table 4.4 compares the performance of three primary hash functions for the FMS model. The functions considered are an ideal random hashing function, $h_1(s) = f_1(s, 7) \bmod r$ and $h_1(s) = f_2(s, 3, 5) \bmod r$. We assume all states are inserted

	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 4.4: 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 when applying three primary hash functions to the states of the FMS model.

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 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.

4.8.3 Secondary Hash Function

For the **secondary hash function**, we consider 32-bit (4-byte) compression based on either f_1 or f_2:

$$h_2(s) = f_1(s, \mathit{shift})$$

or

$$h_2(s) = f_2(s, \mathit{shift}_1, \mathit{shift}_2)$$

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_1 and f_2 can easily be modified to produce a 40-bit hash key instead of a 32-bit hash key.

	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 4.5: The number of unique secondary key values obtained by applying three secondary hash functions to the states of the FMS model.

Table 4.5 compares the performance of the 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 even better than the ideal random hash function.

4.8.4 Hash Function Independence

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 $h_0(s)$.

Table 4.6 shows the correlation between hash function values for various values of k for the states of the FMS model. 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 $h_2(s) = f_2(s,3,5)$. 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. Here the test statistic is given by:

$$t = \frac{r}{s_r} \tag{4.10}$$

where

$$r = \frac{\sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)/n}{(\sum_i x_i^2 - (\sum_i x_i)^2/n)(\sum_i y_i^2 - (\sum_i y_i)^2/n)}$$

denotes the degree of correlation between the values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ $(-1 \le r \le 1)$, and

$$s_r = \sqrt{\frac{1-r^2}{n-2}}$$

is the standard error of r.

The test statistic of Eq. 4.10 t has a t-distribution with $n-2$ degrees of freedom. Using a t-table we can find the two critical probability values corresponding to the given significance level. If t has a value outside of the critical region bounded by these probability values, the null hypothesis that $r = 0$ (i.e. that there is no correlation) will be rejected.

Fig. 4.31 shows scattergrams of the hash function values of 10 000 states sampled 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 4.6. No unusual clusters or patterns are observed in any of the scattergrams. The assumption that our hash

	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$
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 4.6: Correlations between hash function values for the states of the FMS model with $k = 4, 5, 6, 7, 8$.

functions distribute states independently of one another therefore seems to be reasonable in the context of the FMS model.

4.9 Conclusion

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 logarithmic computational cost.

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

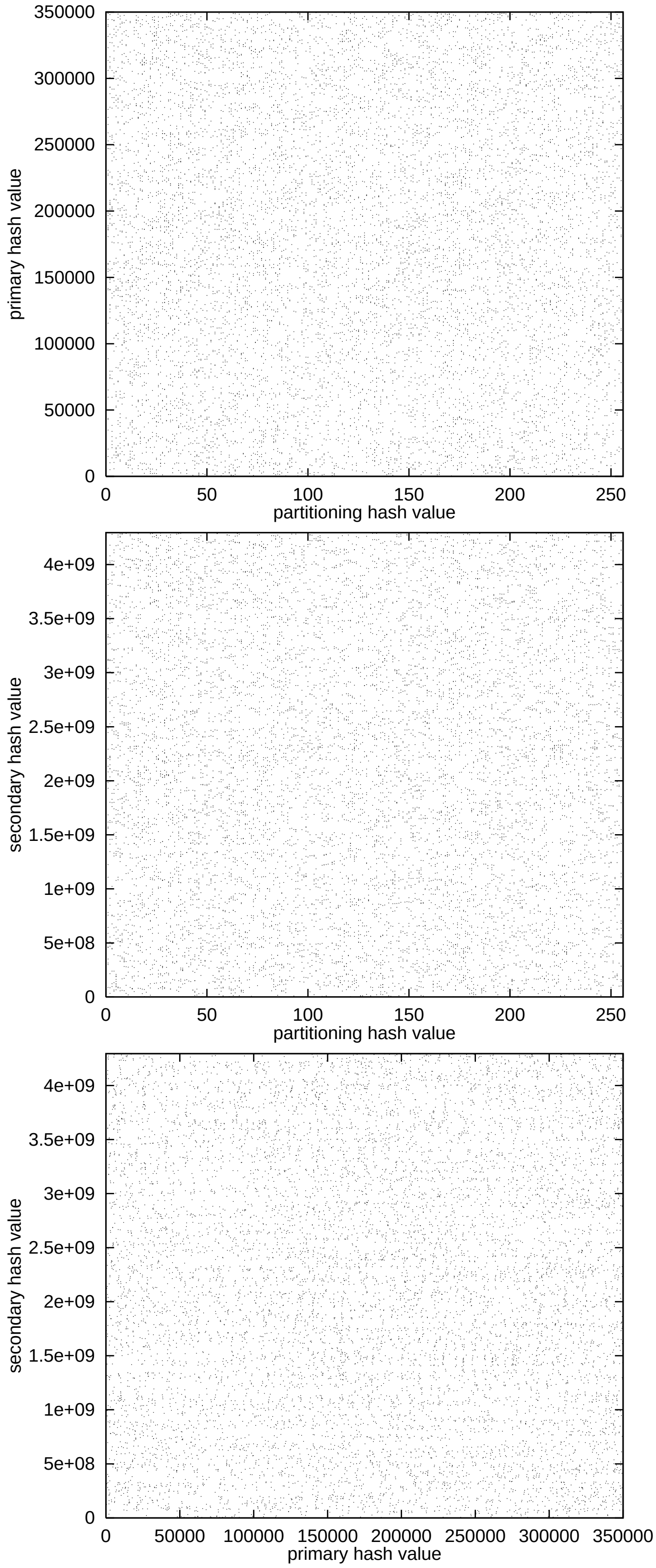


Figure 4.31: Scattergrams of the hash function values of a sample of 10 000 states taken from the state space of the FMS model with $k = 7$.

requires that the hash functions distribute states randomly and independently. We have shown how to generate hashing functions which meet these requirements. To illustrate the potential of our algorithm, we have explored a state space with more than 10^8 tangible states and 10^9 arcs in under an hour using 16 processors on an AP3000 parallel computer. The probability of state omission is less than 0.05%.

Previously, the memory and time bottleneck in the performance analysis pipeline has been state space generation. We believe that our technique shifts this bottleneck away from state space generation and onto stages later in the analysis pipeline. The next chapter will therefore focus on developing a steady-state analyser to solve the state graph's underlying Markov chain for its equilibrium probability distribution.

Chapter 5

A Distributed Disk-based Solution Technique

5.1 Introduction

Having generated the state space and state graph, the next challenge is to find the long run proportion of time the system spends in each of its states. This can be done by mapping the state graph onto a continuous time Markov chain (CTMC) which is then solved for its steady-state distribution.

Solving a CTMC 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 $D = \mathrm{diag}(q^T_{11}, q^T_{22}, \ldots, q^T_{nn})$. π^T is then easily obtained as $D^{-1}y$.

Contemporary methods for the solution of large Markov models were surveyed in Chapter 3. Of these, 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 [DS98a]. Their Block Gauss-Seidel (BGS) solver reads generator matrix elements from disk, maintaining high throughput by using two cooperating processes to 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 focused on methods for distributed and parallel state space generation. Using the techniques presented in Chapter 4, it is now possible to generate very large state spaces of over 100 million states and 1 billion transitions on a 16-node distributed memory parallel computer in about half an hour. 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 (800MB) 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 this chapter, 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 (i.e. the average number of non-zero elements per row) 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 overhead and good load balance.

The kernel is combined 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. The resulting solver has been implemented on a 16-node Fujitsu AP3000 distributed memory parallel computer. We demonstrate the effectiveness of our tool by solving Markov chains of the order of 100 million states and 1 billion transitions.

The rest of this chapter is organised as follows. Section 5.2 outlines the Jacobi and Conjugate Gradient Squared algorithms and considers opportunities for parallelism. Section 5.3 shows how sequential and distributed breadth first generators induce a structure on the generator matrix which can be used to develop an efficient matrix-vector multiply kernel. Section 5.4 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 5.5. Section 5.6 concludes and considers future work.

5.2 Scalable Numerical Methods

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 newly updated steady-state vector elements are used in the calculation of other vector elements within the same iteration. In the case of sparse matrices, this sequential dependency can be alleviated by using multi-coloured ordering schemes which allow parallel computation of unrelated vector elements in phases; however, finding such orderings is a combinatorial problem of exponential complexity. Consequently obtaining suitable orderings for very large matrices is infeasible.

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

Krylov subspace methods [Wei95] (c.f. Section 2.5.2) 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 Krylov subspace algorithms (such as CGS [Son89], BiCGSTAB [Vor92] and TFQMR [Fre93]) are also particularly suited to 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**

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

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

Figure 5.1: The Jacobi method [KGGK94].

5.2.1 Jacobi Method

The Jacobi 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 is the iteration counter (starting 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. 5.1. 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 ($x^{(k)}$, $x^{(k-1)}$ and r).

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

5.2.2 Conjugate Gradient Squared Algorithm

The Conjugate Gradient Squared (CGS) algorithm [Son89] is a generalisation of the classical Conjugate Gradient method (c.f. Fig. 2.7) 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. 5.2. 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). The version we present here is known as the *true residual* version since it explicitly calculates the residual in the last step of every iteration. In our experience, the alternative *updated residual* method exhibits more erratic convergence behaviour since it is prone to the accumulated cancellation effects which occur in finite precision arithmetic.

CGS performs two matrix-vector multiplications, 6 vector updates and 2 vector 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 required for the matrix-vector multiplication, the dot products and the calculation of vector norms involved in the convergence test.

CGS Algorithm

1. Initialise

	1	2	3
• Q^T is given			
• $x^{(0)}$ is an initial guess at the solution		x	
• $r^{(0)} = -Q^T x^{(0)}$	r	x_W	m
• $\hat{r}^{(0)} = r^{(0)}$	r	$\hat{r}_W$	-
• $\rho^{(0)} = 1$			
• $p^{(0)} = q^{(0)} = \underline{0}$	r	p_W	q_W
• $k = 0$			

2. Iterate

	1	2	3
• **while** $\lvert\lvert r^{(k)} \rvert\rvert_\infty / \lvert\lvert x^{(k)} \rvert\rvert_\infty > 10^{-10}$ **do**			
$k = k + 1$			
$\rho^{(k)} = \hat{r}^{(0)} \cdot r^{(k-1)}$	r	$\hat{r}^R$	
$\beta = \rho^{(k)} / \rho^{(k-1)}$			
$u = r^{(k-1)} + \beta q^{(k-1)}$	r	q^R	u
$p^{(k)} = u + \beta(q^{(k-1)} + \beta p^{(k-1)})$	p^R	q	u_W
$v = Q^T p^{(k)}$	p_W	v	m
$\alpha = \rho^{(k)} / (\hat{r}^{(0)} \cdot v)$	$\hat{r}^R$	v	
$q^{(k)} = u - \alpha v$	q	v	u^R
$u = u + q^{(k)}$	q_W		u
$x^{(k)} = x^{(k-1)} + \alpha u$		x^R	u
$r^{(k)} = -Q^T x^{(k)}$	r	x_W	m

3. Normalise x.

Figure 5.2: The CGS algorithm [Son89].

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 matrix-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. 5.2. The numbered columns reflect the contents of the n-vectors at each step of the algorithm. 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.

5.3 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 cost, 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 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 efficient matrix-vector multiply kernel.

5.3.1 Infinitesimal Generator Matrix Structure

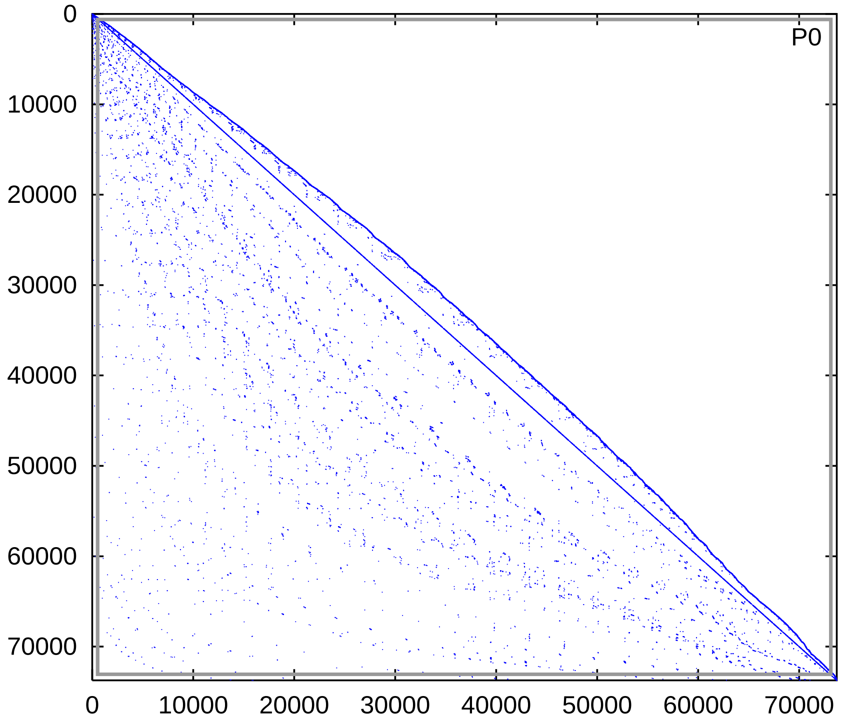
Automated state space generation tools are widely used to map structurally unrestricted high-level models onto their underlying state spaces and generator matrices. Typically this mapping is performed using a sequential breadth-first search (BFS) traversal which assigns unique state sequence numbers to states in the order in which they are encountered. Fig. 5.3(a) demonstrates how a sequential BFS generator induces a (mostly) lower-triangular structure on the resulting generator matrix Q. The matrices shown in the figure 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.

Distributed state space generators (e.g. [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, the ith 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 5.3(b) illustrates the resulting structure induced by a distributed BFS generator.

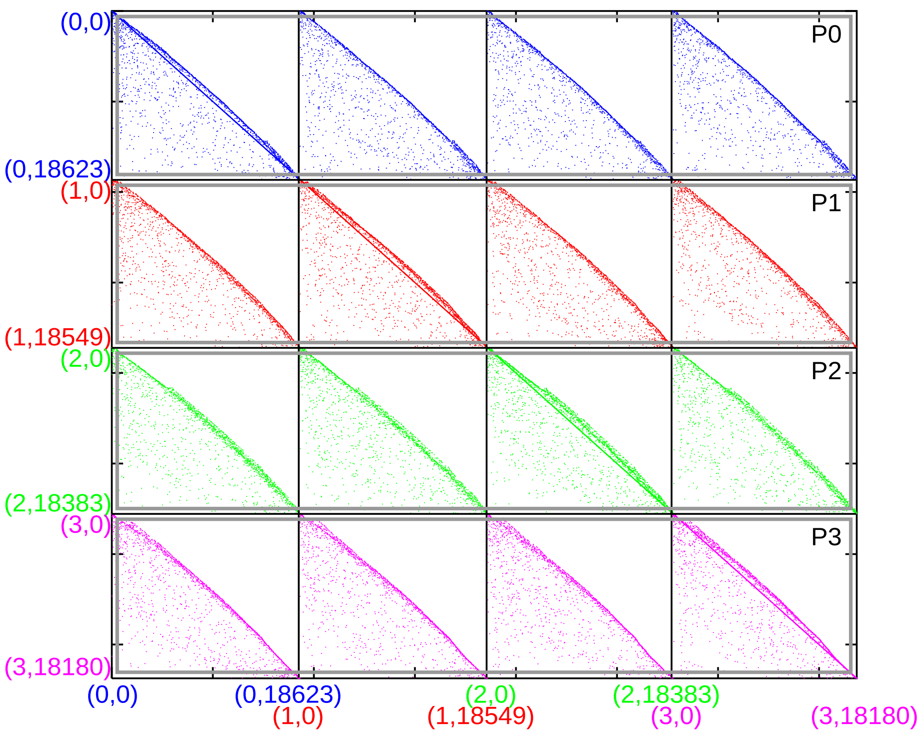
5.3.2 Matrix Reordering Strategies

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 matri-

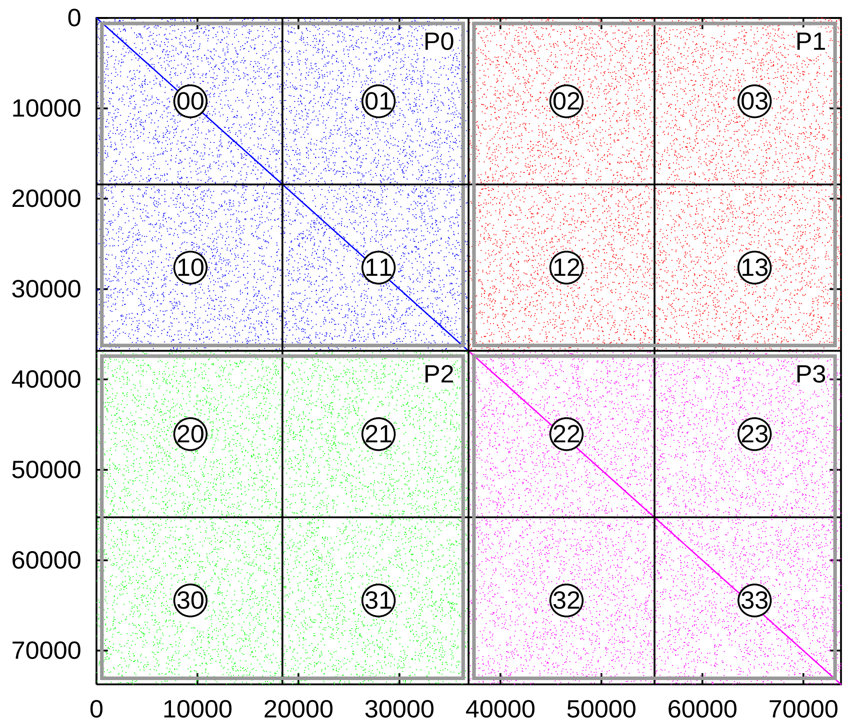


(a) Sequential BFS generator.

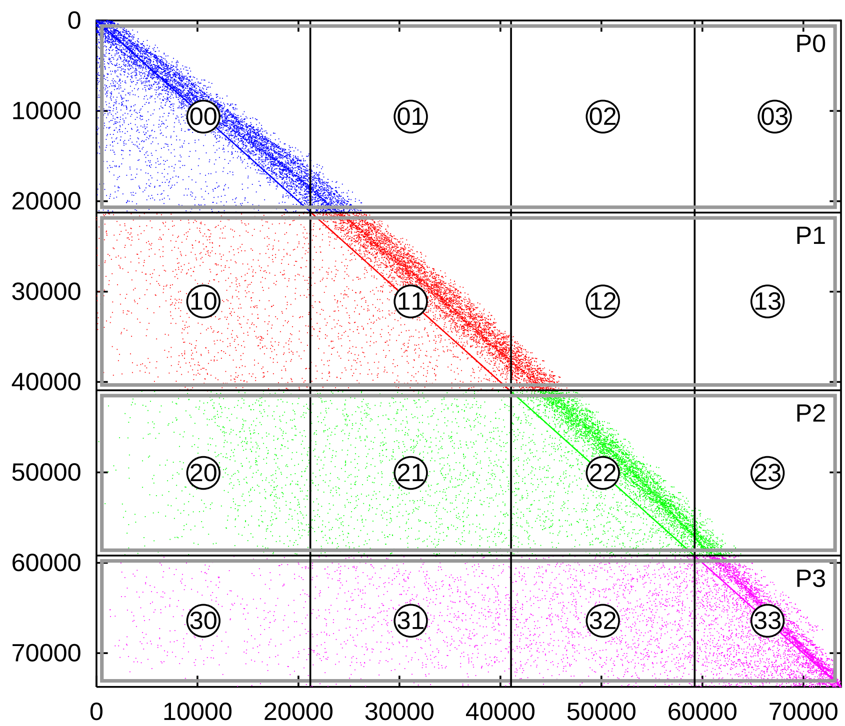


(b) Distributed BFS generator (4 processors).

Figure 5.3: The non-zero structure of the generator matrix Q as induced by sequential and distributed breadth-first state generators.



(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 5.4: The structure of Q^T after state reordering and corresponding processor assignments.

ces 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 parallel sparse matrix-vector multiplication [LG93, LPG94, HLP95].

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

$$f(i,j) = (c_1 * (\sum_{k=0}^{i-1} n_k + j) + c_2) \bmod n \tag{5.1}$$

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. 5.4(a) shows the resulting layout of Q^T after the application of this mapping and a corresponding block-checkerboard assignment of processors.

Approximate BFS reordering

The random remapping described above allows for the application of well-known efficient matrix-vector multiplication algorithms for unstructured matrices, but these suffer 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. 5.3(b). In particular, if we assign state (i, j) to a global state number given by the function

$$f(i,j) = n - \sum_{k=0}^{i} \min(j+1, n_k) - \sum_{k=i+1}^{p-1} \min(j, n_k) \tag{5.2}$$

we obtain the BFS-like structure for Q^T shown in Fig. 5.4(b). It is then a simple task to assign blocks of consecutive states to processors such that the number of non-zeros allocated to each processor is the same. This results in a row-wise allocation of matrix-blocks to processors, as shown in Fig. 5.4(b). Note how this mapping results in more rows being allocated to the lower-numbered processors on account of the lower-triangular structure of the generator matrix.

5.3.3 Kernel Algorithm

We now outline a disk-based distributed sparse matrix-vector 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.

The algorithm for node i which performs $y_i = Q^T_{i*}x_i$ is outlined in Fig. 5.5. 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 range 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,

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
 $y = \underline{0}$
 for $k = 0$ **to** $p - 1$ **do begin**
  $j = (i + k) \bmod p$
  if Q^T_{ij} is empty **continue**
  if $i \neq j$ **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 5.5: Distributed sparse matrix-vector multiply kernel for node i.

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 achieving the complete overlap of communication and computation.

5.4 Distributed Disk-based Solver 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. 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 [DS98a] (c.f. Section 3.2.5). However, in a parallel context, it is also important to overlap communication with disk I/O. We therefore propose a distributed disk-based architecture that has the added benefit of allowing communication to proceed in parallel with disk I/O.

Fig. 5.6 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

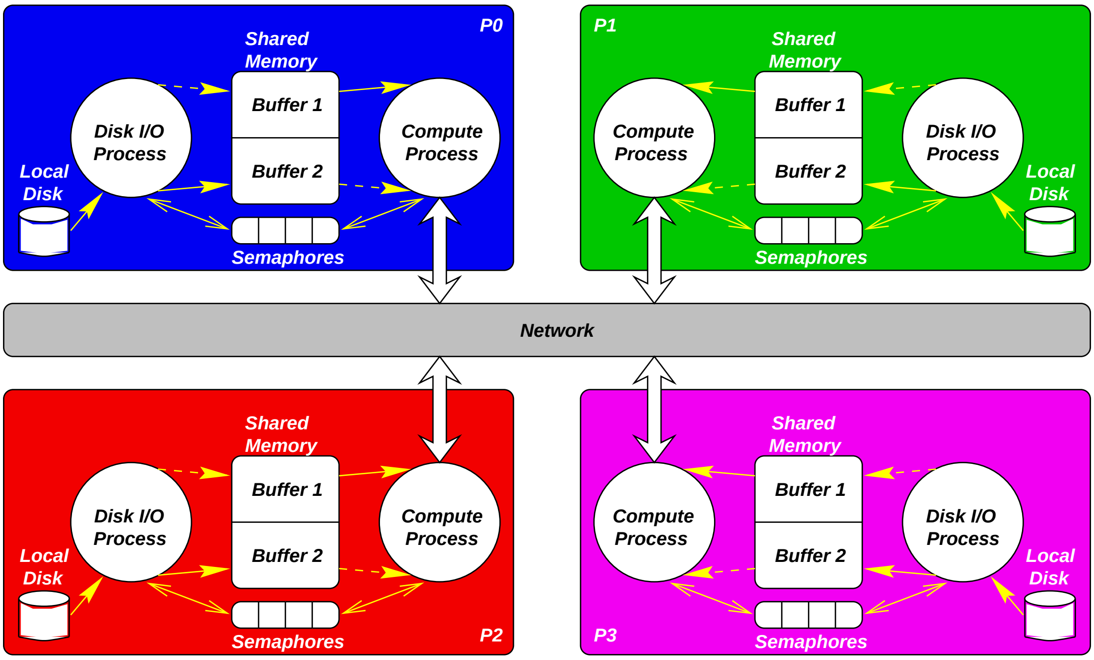


Figure 5.6: Distributed disk-based solver architecture

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. The compute process also manages interprocess communication. Interested readers are invited to consult Section 6.2.4 for more technical details about this architecture and how it can be realised in practice.

5.5 Results

We have implemented a distributed disk-based Markov solver which uses the kernel described in Section 5.3.3 and the software architecture outlined in Section 5.4. 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 collected using up to 16 processors on the AP3000. Further details of the implementation are given in Chapter 6 while full details of the machine and its configuration are given in Appendix B.

5.5.1 The FMS Model

The first example we consider is a 22-place GSPN model of a flexible manufacturing system (the FMS model). A full description of this model is given in Appendix A.2. Fig. 4.10 shows the number of states and transitions in the model in terms of the parameter k (which corresponds to the initial number of parts in the system). We use the same transition rates (many of them state-dependent) as given in the original model specification [CT93].

The state spaces and generator matrices for the models were generated using the distributed state generation algorithm presented in Chapter 4. The generator matrices were then transposed and remapped in a distributed fashion according to the reordering presented in Section 5.3.2. 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 30 minutes 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 an 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 (i.e. the average number of non-zero elements in a row) 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 5.1 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

		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)	20.9	23.8	33.4	61.0	131.5			
$p = 2$	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)	20.5	21.9	27.00	41.3	78.0			
$p = 4$	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)	20.3	21.1	23.8	31.5	51.2			
$p = 8$	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)	20.1	20.6	22.2	26.6	37.8	64.2		
$p = 12$	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)	20.1	20.5	21.6	24.9	33.4	53.2	96.2	
$p = 16$	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)	20.1	20.4	21.3	24.1	31.1	47.6	83.5	102.0

Table 5.1: Real time in seconds required for the distributed solution of the FMS model.

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. 5.7 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-vector 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 values of k, with the CGS algorithm typically converging about 4 times faster than the Jacobi method.

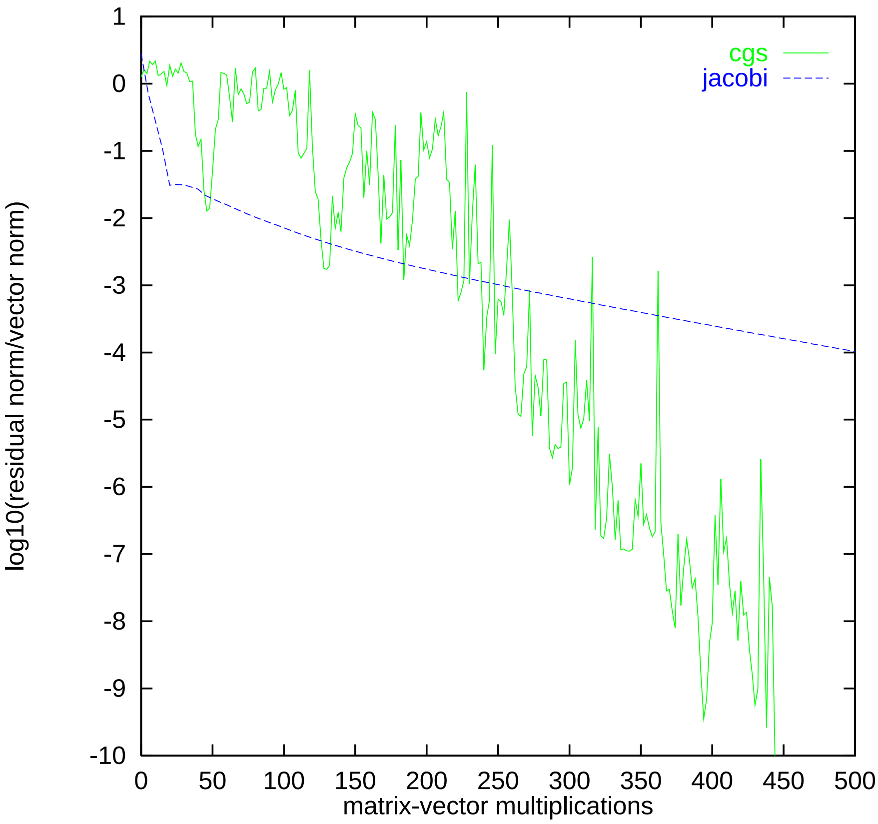


Figure 5.7: Jacobi and CGS convergence behaviour for the FMS model with $k = 7$.

The largest model that can be solved on a single processor is the case $k = 8$ (4.5 million states). Fig. 5.8 compares the average time taken per distributed CGS iteration for model sizes up to $k = 8$ using various numbers of processors.

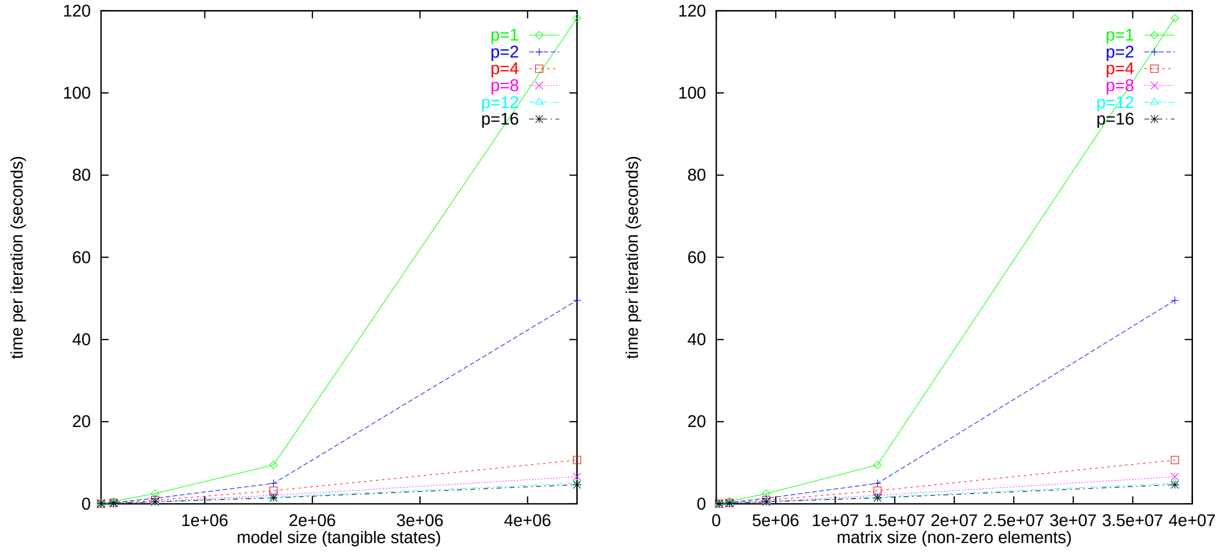


Figure 5.8: Time per distributed CGS iteration in seconds relative to the number of states (left) and number of transitions (right) in the FMS model.

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. 5.9 shows the speedup and efficiency achieved by the CGS method for small FMS models with $k \leq 7$ where variation due to caching effects does not play an important role. The case $k = 8$ gives superlinear speedup due to caching effects and is not shown. The speedup S_p for p processors is given by the run time of a sequential CGS iteration ($p = 1$) divided by the run time of a distributed CGS iteration with p processors. Efficiency for p processors is given by S_p/p. We see that larger problem sizes produce better speedups, and that adding processors increases the speedup for all but the smallest problems where communication costs dominate. The efficiency graph shows that efficiency decreases as we add more processors, as is to be expected when the problem size is maintained at the same level. Note that, although the efficiencies reported here are lower than those observed for the state space generation algorithm due to the nature of the problem, they are competitive with the speedups reported in the literature for parallel linear equation solvers (c.f. Section 3.2.6).

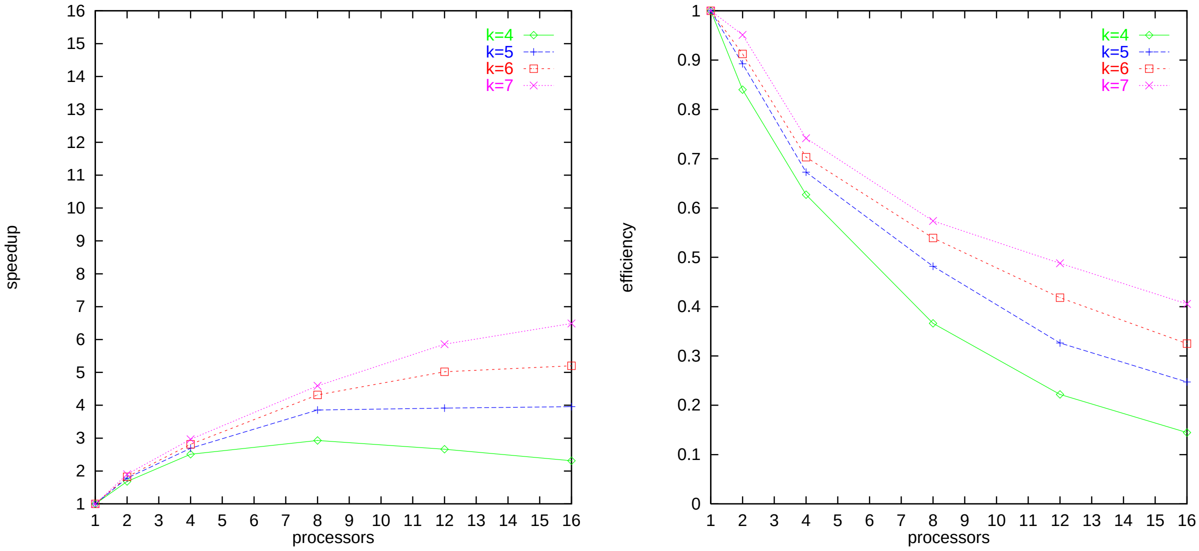


Figure 5.9: CGS speedup (left) and efficiency (right) for the FMS model with $k = 4, 5, 6, 7$.

Moving beyond the maximum problem size that can be handled 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 1.5TB, with the nodes jointly processing an average of 144MB of disk data every second.

The case $k = 11$ (54 million states) is solved in 17 hours 20 minutes on 16 processors. Here the total amount of disk I/O required is over 4TB, with the nodes jointly processing an average of 67MB of disk data every second.

We note that the main obstacles to solving the largest state space generated ($k = 12$) is a practical limit imposed by insufficient shared disk space (10GB available). Shared disk space is used to store the state space, the state graph and the remapped transposed generator matrix. The peak requirement for this space occurs during matrix transposition and remapping, and is about 13GB in this case. For an estimated 450 CGS iterations, over 10TB of disk I/O would be needed, requiring nearly 3 days of processing time if the 16 nodes sustain an average of 48MB of disk data per second (i.e. 3MB/s each). Approximately

187MB memory per node would be required.

5.5.2 The Courier Protocol Model

The second example we consider is a 45-place GSPN model of the Courier telecommunications protocol software. This model is fully described in Appendix A.3. Like the FMS model, the Courier model has a scalable parameter k (corresponding to the sliding window size). Fig. 4.19 shows the number of states and transitions in the model in terms of k.

As for the FMS model, the state spaces and generator matrices for the models were generated using the distributed state generation algorithm of Chapter 4, and remapped according to the approximate BFS ordering of Section 5.3.2. The generation and remapping are very rapid – the $k = 7$ (40 million states) case can be generated and remapped in under 20 minutes of real time.

Table 5.2 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 $k = 1$ (11 700 states) to $k = 7$ (39.8 million states) and runs are conducted on 1, 2, 4, 8, 12 and 16 processors. The number of iterations for convergence and memory use per processor are also shown.

Fig. 5.10 compares the convergence of the Jacobi method with that of the CGS algorithm for the $k = 4$ case in terms of the number of matrix-multiplications performed. As was the case for the FMS model, the Jacobi method begins by converging quickly, but then plateaus, converging very slowly but smoothly. The CGS algorithm, on the other hand, exhibits erratic rapid convergence that improves in a concave fashion.

The largest model that can be solved on a single processor is the case $k = 6$ (5.4 million states). Fig. 5.11 compares the average time taken per distributed CGS iteration for model sizes up to $k = 6$ using various numbers of processors. Again, there is a dramatic reduction in run-time as processors are added; this effect can

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

Table 5.2: Real time in seconds required for the distributed solution of the Courier model.

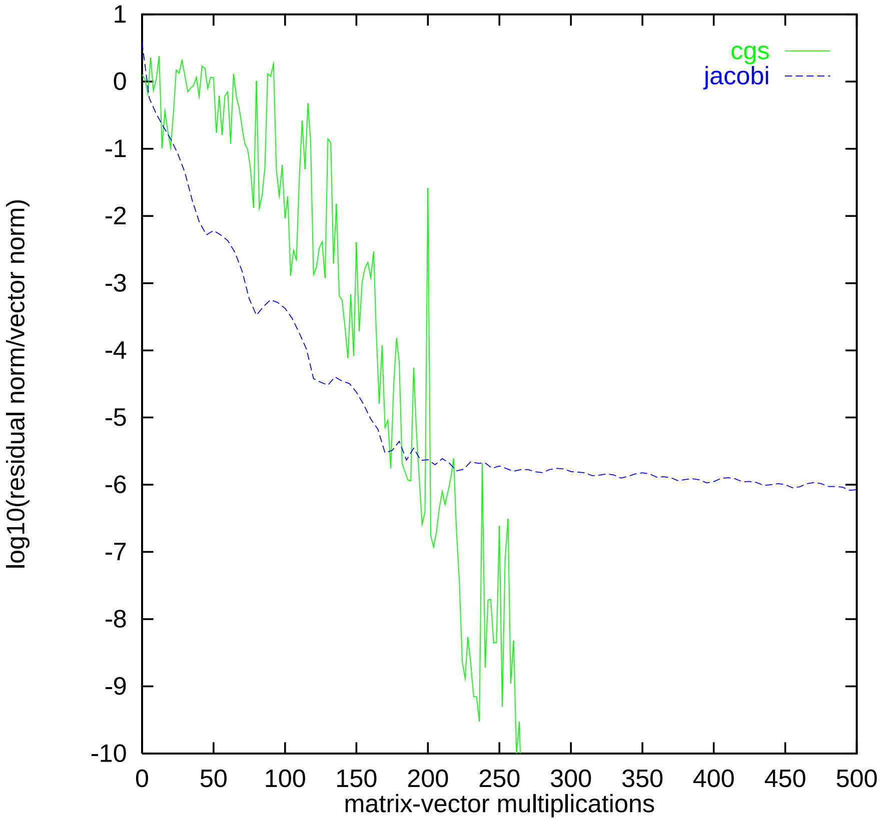


Figure 5.10: Jacobi and CGS convergence behaviour for the Courier model with $k = 4$.

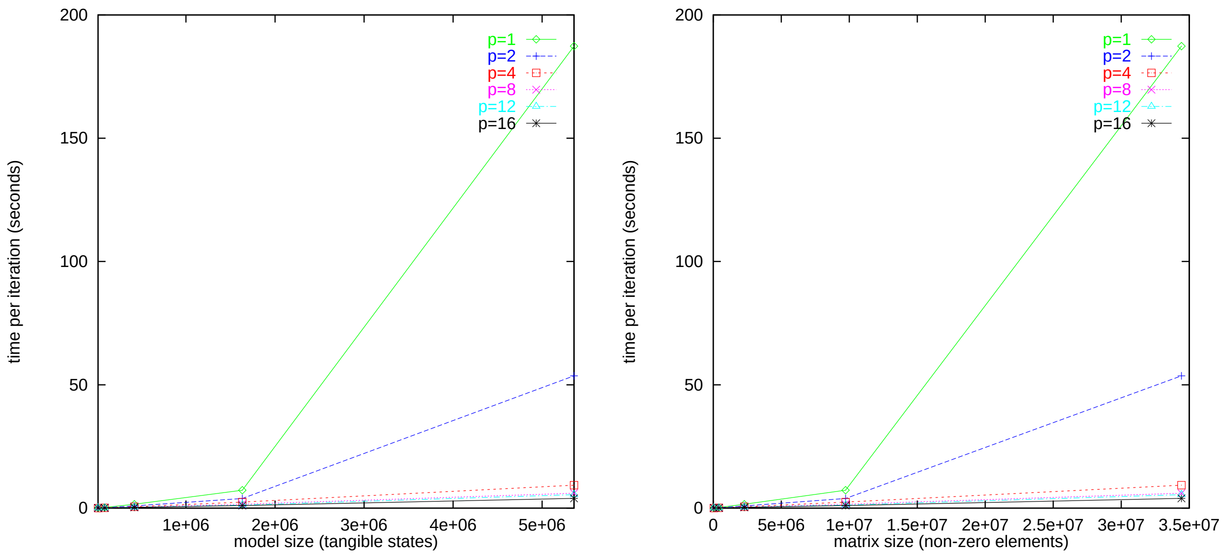


Figure 5.11: Time per distributed CGS iteration in seconds relative to the number of states (left) and number of transitions (right) in the Courier model.

be attributed to better caching of disk I/O as processors are added.

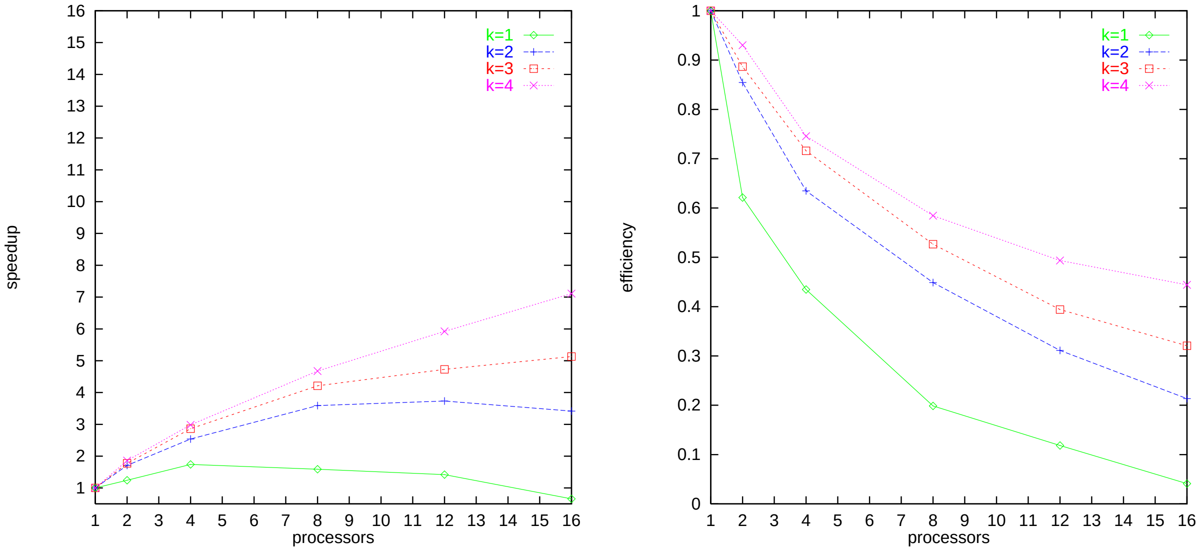


Figure 5.12: CGS speedup (left) and efficiency (right) for the Courier model with $k = 1, 2, 3, 4$.

Fig. 5.12 shows the corresponding speedup and efficiency achieved by the CGS method for small Courier models with $k < 4$. As we observed for the FMS model, larger problem sizes produce better speedups. Adding processors generally increases the speedup for a given problem size, although at some point communication costs begin to dominate. For the case $k = 1$ with 16 processors, there is even a slowdown over the single processor case because of the high communications overhead. It would be an interesting challenge to derive a theoretical performance model which determines the number of processors required to produce an optimal speedup for a given problem size, although the derivation of such a model would be complicated by the need to account for the effect of disk caching. The corresponding efficiency graph shows that efficiency decreases as we add more processors, as is to be expected when the problem size is maintained at the same level.

Moving beyond the maximum problem size that can be handled on a single processor, the $k = 6$ case (15 million states) is solved in under 34 minutes using 16 processors, while the $k = 7$ case (40 million states) is solved in 3 hours 50 minutes. 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 83MB of disk data every second.

The largest state space solved is the $k = 8$ case (94 million states) which takes 1 day 15 hours of processing time on 16 processors. The total amount of I/O across all nodes is 3.4TB, with the nodes jointly processing an average of 24MB disk data every second. This disk throughput of only 1.5MB/s per node is two to three times lower than would normally be expected. Investigation revealed that the poor performance is caused by high virtual memory paging activity. This suggests that there is insufficient physical memory, even though the solver data and program have a theoretical per node memory requirement of only 161MB (while each node has 256MB RAM). Further inspection showed that real per node memory use is in fact much higher, since the MPI library uses 50MB for its communication buffers (in this case the longest messages sent between processors are 47MB long) and the operating system requires about 50MB of resident code, essential system daemons etc. One way to reduce the program's working set of over 260MB so that it fits entirely within the available memory would be to divide long messages into several smaller chunks, thus decreasing the size of communication buffers used by the MPI library.

5.6 Conclusion

This chapter has presented a distributed disk-based method for solving very large Markov chains. We have selected appropriate numerical methods and investigated the structure of generator matrices produced by sequential and distributed breadth-first state space generators. This structure has been exploited 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 also described a software architecture for a distributed disk-based Markov solver. This software architecture uses two processes per node which allows disk

I/O to proceed concurrently with computation and communication. The solver has been implemented on a 16-node distributed memory parallel computer and used to solve models with of the order of 100 million states and 1 billion 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 needed to store the solution vector alone would be over 800MB.

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.

Chapter 6

A Parallel Performance Analysis Pipeline

6.1 Introduction

This chapter places the work of Chapters 4 and 5 in context by describing an implementation of a complete parallel performance analysis pipeline. The pipeline accepts a high-level description of a dynamic performance model, generates the underlying state space and state graph, derives and solves the steady-state equations of a Markov process and produces performance statistics. The modules in the pipeline have been implemented in C++ using the Message Passing Interface (MPI) standard, so the analysis pipeline can be ported to a wide range of parallel computers and high-performance workstation clusters.

Section 6.2 gives an overview of the complete analysis pipeline. Each major module is then described in detail, concentrating on more interesting implementation aspects such as data structures, algorithms, and the results of various optimization experiments.

The generality and unrestricted nature of our approach poses the problem of developing an interface language which is powerful enough to specify a wide variety of stochastic models. Section 6.3 presents such a high-level interface language while Section 6.4 presents a sample system specification.

6.2 Analyser Components

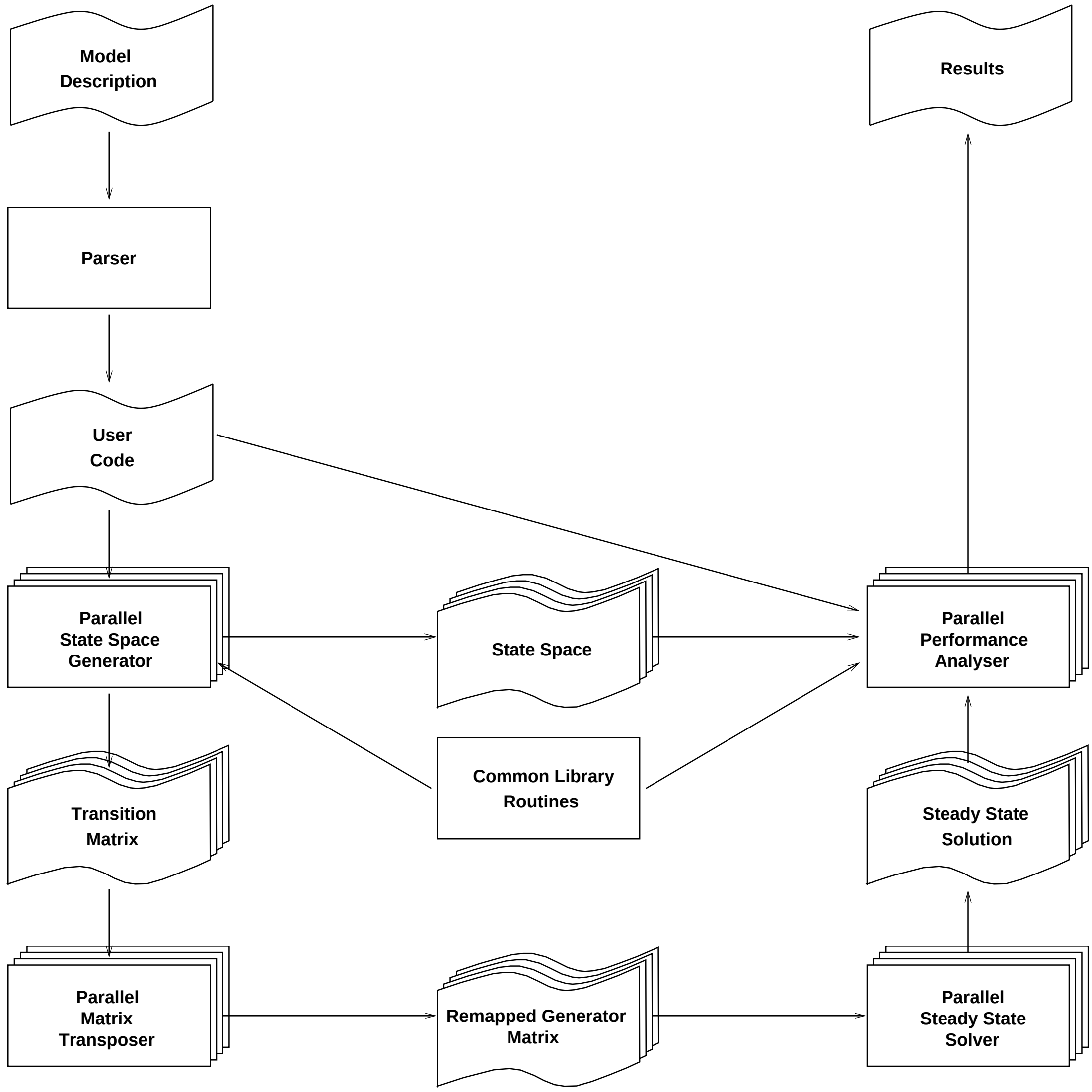


Figure 6.1: Main modules of the parallel performance analysis pipeline

Fig. 6.1 illustrates the major modules of the analysis pipeline. Control is passed from module to module as follows:

- The **parser** translates a high-level model description (specified using the interface language of Section 6.3) into a C++ class describing the same model. We refer to the generated C++ class as "user code".

- The user code is compiled and linked with common library routines to form a **parallel state space generator** for the model. The state space generator uses the parallel probabilistic algorithm described in Chapter 4 to produce the distributed state space and distributed transition matrix.

- A **parallel matrix transposer** remaps and rebalances the transition matrix (as described in Section 5.3.2) to derive the generator matrix required by the steady-state solution process.

- Next the **parallel steady-state solver** determines the stationary distribution of the Markov chain by solving the large set of linear equations derived from the generator matrix. The solver makes use of the distributed disk-based kernel described in Chapter 5.

- Finally, the user code is linked with common library routines to form a **parallel performance analyser**. The performance analyser makes use of the steady-state solution as well as the state space to produce performance results.

The following sections consider each component in turn and describe some of the more interesting implementation details.

6.2.1 The Parser

A simple recursive descent parser [ASU86] implements an interface language for specifying a wide variety of high-level models (c.f. Section 6.3). The parser accepts a user data file containing:

- A **model description** including the format of the state descriptor, an initial state and rules governing transitions between states.

- A description of **performance statistics** to be computed in the form of state or count measures.

- User **options** relating to the state space generation and steady-state solution, such as desired accuracy or choice of solution method.

If there are no syntatic errors, the backend of the parser generates the user code necessary for state space exploration and performance analysis. The user code is encapsulated in a C++ `State` class so that every model presents a uniform high-level interface to external program modules such as the state space generator and the performance analyser. In particular, the generated `State` class includes methods to:

- Set up the current state as the initial state specified for the system.
- Determine the sets of timed and immediate transitions enabled by the current state.
- Fire any enabled timed or immediate transition to determine the successor state.
- Determine the (possibly state-dependent) rate or weight of any transition.
- Calculate the partitioning, primary and secondary hash keys for the current state, as required by the parallel probabilistic state exploration algorithm.
- Check that user-specified invariants apply to the current state.
- Compute performance statistics for the current state in the form of state and count measures.

Since this high-level interface does not change from model to model, the relatively small amount of code found in the model-specific `State` class can be compiled and linked with precompiled library code to produce a parallel state generator and a parallel performance analyser for the model. This reduces compilation time considerably.

6.2.2 The Parallel State Space Generator

The state space generation algorithm of Chapter 4 has been implemented in C++ using the Message Passing Interface (MPI) model of parallel computation. Under this model, there is a fixed pool of N worker processes, each of which executes a copy of the same code[1]. The behaviour of the workers is not identical, however, since MPI allows worker processes to distinguish their behaviour from that of the others based on their identity or *rank*.

As shown in Fig. 6.2, each worker process makes use of seven main data structures:

- A **state hash table** used to store and search for local states based on their primary and secondary hash keys. The hash table has 750 019 rows and uses 40-bit secondary keys. The rows are implemented as dynamic arrays.
- A **FIFO state queue** used to store unexplored tangible states. This queue can grow to be very large, but is accessed in a predictable fashion. Consequently, the queue is divided into pages and managed so that only the front and back pages of the queue are kept in memory, while the main body of the queue is held on disk.
- A **vanishing state stack** used for the temporary storage of states during elimination of vanishing states. The on-the-fly procedure used by the analyser to eliminate states in which the system spends no time leads to a decrease in the number of states generated, memory usage and transition matrix size. Interested readers can consult [CMT91] and [Kno96] for further details.
- A set of **message buffers** used to implement efficient communication. Outgoing hash keys, states and identity messages are held in these buffers according to their destination. When the size of a message buffer corresponding to a particular destination exceeds a threshold, the messages in that

[1]Strictly speaking this applies only to the MPI-1 standard, since the recent MPI-2 standard allows for the dynamic creation and destruction of processes.

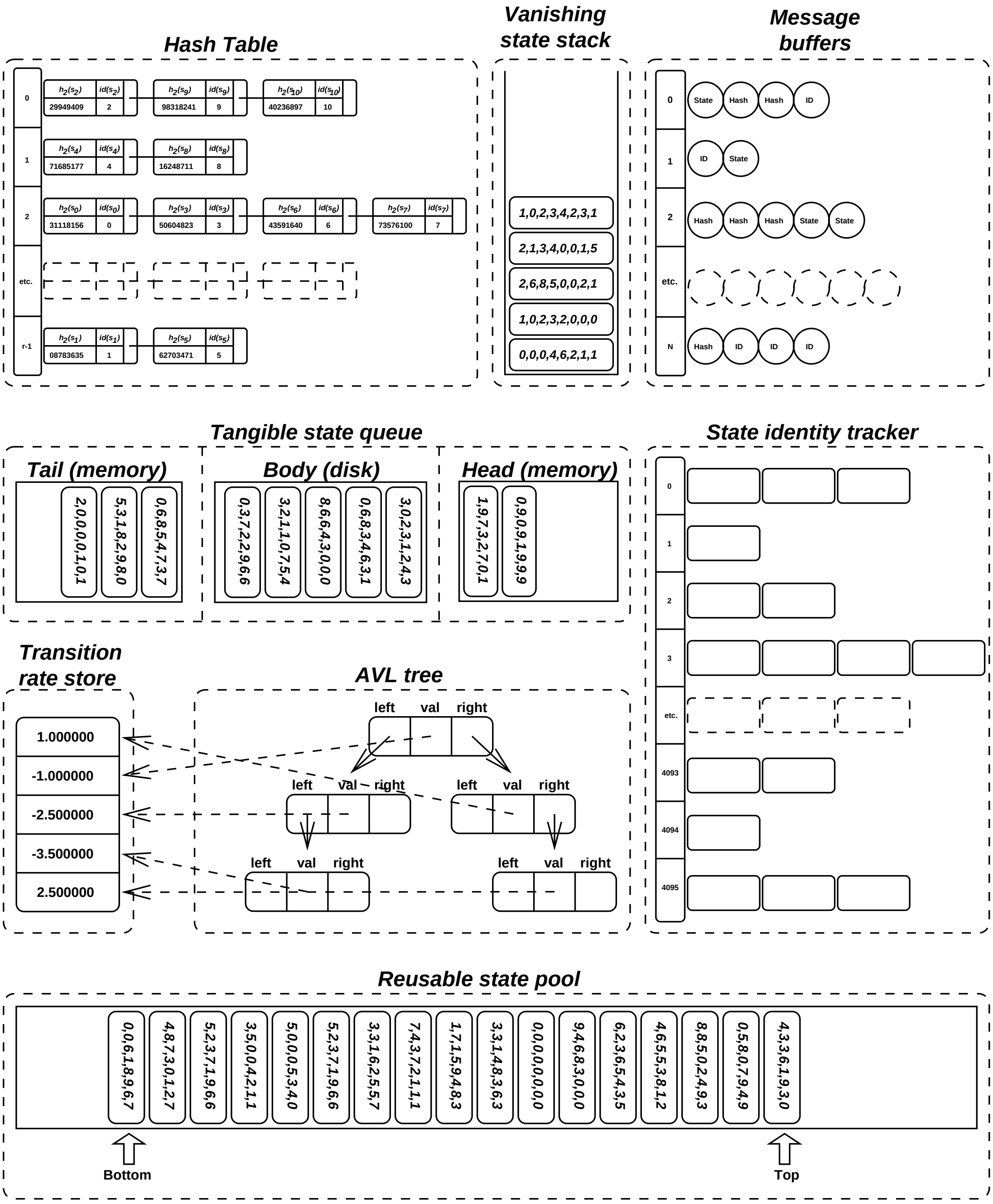


Figure 6.2: Data structures used by each worker process of the parallel state generator

buffer are packed into a large block and sent. This prevents the network from being flooded by many short messages, but introduces the possibility of starvation or deadlock. To prevent this, processors that have very few states left in their FIFO queue broadcast a message to other nodes requesting them to flush their outgoing message buffers.

- A **state identity tracker** used to store states while the state sequence numbers of their successor states are established. The tracker takes the form of a hash table with a small number of slots. Parent states are allocated a free slot in the tracker until all their children have been identified by the processors that own them. At this point a complete row of the transition matrix is written to disk and the slot is freed. The number of slots in the tracker varies according to network conditions, with a maximum size of 4096 slots. If the number of slots required exceeds this maximum threshold, new state exploration activity is temporarily suspended and a message is broadcast to all other nodes requesting them to immediately send any messages waiting in their state identity queues. By flushing their identity queues, the nodes receiving the request return to the sender *all* outstanding identities. This frees up all the slots in the sender's identity tracker and allows state space exploration activity to continue as normal.

- An **AVL tree** (height-balanced binary tree) used as an index into a vector of common transition rates. Since transition rates are often fixed and are seldom state-dependent, models typically have few distinct transition rates. Thus, instead of storing a double precision floating point number to denote each rate, it is more economical to store an index into a vector of common transition rates. As entries are added to the transition matrix, a rapid search mechanism is needed to establish whether an entry is already in the store. An AVL tree is thus maintained to rapidly search for store items, reducing the search complexity from $O(n)$ for a linear search of the store to $O(\log_2 n)$.

- A **reusuable state pool** used to enhance the efficiency of memory allocation. The exploration process involves the continuous creation of child states and the destruction of explored states. By placing discarded states in a circular buffer and recycling them instead of creating new states, the overhead of operating system memory allocation calls can be avoided. If the pool floods, or if there is an underflow, operating system allocation calls are used as usual. The result is a dramatic improvement in run time of up to 50%.

The main core of the state explorer implements both the original algorithm of Fig. 4.5 as well as the communication efficient version of Fig. 4.7. Both algorithms include facilities for checking invariants on every state as soon as they are generated. This helps to detect unsafe or unexpected configurations caused by specification errors.

In order to save disk space, the exploration algorithm writes the state space and state graph to disk in compressed format. To find a good compression method, we compared the compression ratio and encoding speed of three algorithms, viz. our own LZ77 compression technique, the deflation algorithm (LZ77/Huffmann combination) used by the `gzip` utility and provided by the `zlib` compression library[2], and the LZW encoding provided by the UNIX utility `compress` [Sal98]. The UNIX `compress` utility was found to provide the fastest encoding speed and competitive compression ratios, while the `zlib` routines provided the best compression ratios but the worst encoding speed. Consequently, compression is implemented very simply by piping data directly to the `compress` utility using the `popen` (open pipe) system call. Compression activity consumes about 15% of the CPU during state exploration.

Experiments were also conducted to study the effect of introducing identity caches on each node. The caches reduce the amount of communication needed to identify child states by using a hash table of fixed size to store the identities of remote states as they are received. Messages requesting the identities of child states are

[2] available from `http://www.cdrom.com/pub/infozip/zlib/`

only sent if the relevant identities are not already present in the cache. While this scheme was found to be effective for small numbers of nodes, it did not scale to a large number of nodes. This is because the hit rate drops off as processors are added, until eventually it is too low to compensate for the overhead of cache lookup. Consequently, the final version of the state generator does not use this data structure.

6.2.3 The Parallel Matrix Transposer

The output from the state space generator is the state space and a distributed transition matrix R that corresponds to the state graph. The next step in the analysis pipeline is to transform the distributed transition matrix R into the distributed transposed generator matrix Q^T. Q^T is required since the steady-state solution module solves a set of linear equations of form $Ax = b$ with $A = Q^T$, $x = \pi^T$ and $b = \underline{0}$. The transposer also helps to improve the efficiency of subsequent parallel matrix-vector multiplication operations in two ways. Firstly, the rows and columns of Q^T are permuted to give the matrix a structure that reduces the amount of communication required during matrix-vector multiplication. Secondly, non-zero entries in Q^T are balanced across processors to give each processor a similar workload.

In the following, we will assume that the ith of N processors has generated n_i states of the state space ($0 \le i < N$) and also the corresponding n_i rows of the state graph which we denote by R_i. The total number of states generated is $n = \sum_{i=0}^{N-1} n_i$. We further assume that the state generator has used identifiers of form (i, j) to label states, where i is the processor number ($0 \le i < N$) and j is the local state sequence number ($0 \le j < n_i$). A transition between states (i, a) and (j, b) at rate μ is denoted by the transition matrix entry $(i, a) \xrightarrow{\mu} (j, b)$.

Remapping, rebalancing and normalisation

A function $\texttt{remap}(i,j) \rightarrow (0,1,\ldots,n-1)$ is used to map state identifiers of form (i,j) onto a global state number. This mapping reorders the rows and columns of the transition matrix, using either approximate breadth-first search reordering (c.f. Eq. 5.2) or random reordering (c.f. Eq. 5.1).

Another function $\texttt{owner}(k) \rightarrow (0,1,\ldots,N-1)$ is used to map global state k (and the corresponding rows of Q^T) onto a host processor. Initially this mapping is based on a straightforward allocation of n/N states per processor.

Following the initialization of the `remap` and `owner` functions, the ith processor scans the entries in R_i and compiles summary statistics about the distribution of states that would result from a transposition and remapping. These statistics are used to adjust the assignment of states to processors given by the `owner` function so that the number of non-zeros in the rows of Q^T allocated to each processor is the same.

As the scan proceeds the row sums of R are calculated and stored in a distributed vector `sum`. For each row of R corresponding to transitions from state (i,j), the row sum is stored in the vector element $\texttt{sum}[\texttt{remap}(i,j)]$. This vector will be used to normalise the rows of R such that all diagonal entries in Q^T are -1. For an explanation of why it is safe to perform this normalisation, see Section 5.1.

Transposition

Having refined the `owner` function and calculated the `sum` vector, we are in a position to carry out the transposition of R subject to the new state mapping and normalisation of the transition rates.

Each processor i now scans the entries in R_i for a second time. Each non-zero entry in R_i of form

$$(i,a) \xrightarrow{\mu} (j,b)$$

results in the transmission of the transposed and normalised matrix entry

$$\texttt{remap}(j,b) \xrightarrow{\mu/\texttt{sum}[\texttt{remap}(i,a)]} \texttt{remap}(i,a)$$

to processor $\texttt{owner}(\texttt{remap}(j, b))$. Outgoing matrix entries are buffered according to destination and sent in large blocks to prevent the network from being flooded by many short messages.

Each processor stores the matrix entries it receives on disk. Storage of the entries on each processor is divided into N files according to row index such that the jth file on processor i corresponds to entries in matrix block Q_{ij}^T. These blocks contain entries of form $a \xrightarrow{\mu} b$ where a and b are global state numbers (with $\texttt{owner}(a) = i$) and μ is a normalised transition rate. To manipulate these entries into a more compact sparse matrix format, each block is sorted by row index and written to disk in three separate files – one for the column indices, one for transition rates and one for the number of non-zero entries in each row of the block. Four bytes are required to store the column indices. As for the state generator, an AVL tree and rate vector are used to store transition rates so only 2 bytes are needed for each rate instead of the 8 that would be needed for a double precision floating point number. One byte is used to store the number of non-zero entries in each row of the block. These disk files are now prepared for the next phase, which is the disk-based steady-state solution process.

6.2.4 The Parallel Steady-state Solver

A parallel steady-state solver written in C++ implements the sparse matrix-vector multiplication kernel algorithm of Fig. 5.5 in the context of the distributed disk-based architecture of Fig. 5.6. Like the generator and the transposer, the solver uses MPI so at startup time there are N worker processes (one per node). Each worker or master process then creates a slave process using the `fork()` system call.

Master and slave communicate by using standard System V interprocess communication (IPC) library calls to attach themselves to two shared memory buffers used to channel data and to a set of semaphores used for synchronization purposes. The slave process acts as a data producer, reading the data files prepared by the matrix transposer and filling one of the shared memory buffers. While one shared memory buffer is filled, the master process consumes the data in the other buffer. The slave process only handles disk I/O, whereas the master process is responsible for all computation and communication with other master processes. In this way maximum disk throughput is achieved through the overlap of disk I/O with communication and computation.

To save memory, all vectors used by the solver, including the steady-state vector, are completely distributed. Since matrix blocks contain different numbers of non-zero elements and must be multiplied with portions of the steady-state vector owned by other processes, there is a need for nodes to be able to process remote requests for subvectors during computation. Ideally this should be handled by using a dedicated thread to process subvector requests. However, the MPI library we use is not thread-safe, so `MPI_Iprobe()` non-blocking probe operations are used to periodically check for incoming subvector requests.

Optimizations

The solver includes some options to improve the efficiency of the matrix-vector multiplication kernel at the cost of extra memory:

- The kernel algorithm blocks while waiting for remote subvectors to be arrive. At the cost of an extra distributed solution vector, the non-blocking communication primitives `MPI_Isend()` and `MPI_Irecv()` can be used to receive subvectors required by subsequent blocks while the computation with the current block proceeds.

  Surprisingly, this optimization did not result in a significant performance improvement when implemented on the Fujitsu AP3000 distributed memory parallel computer. Followup investigations revealed two deficiencies in Fujitsu's MPI implementation and hardware architecture: the "non-blocking" `MPI_Isend()` call in fact blocks for the same length of time as a blocking send, and communication requires such large amounts of CPU overhead that virtually no user instructions are executed while data is actively being sent or received (see Appendix B.4.2 for a more complete description of this problem). We anticipate, however, that this would a worthwhile optimization on parallel machines that have more effective support for the overlap of communication and computation (such as the IBM SP-2).

- By default, the number of non-zero entries in each row of each (non-empty) generator matrix block is read in from disk during each iteration. Since only one byte is required to store the number of non-zero entries in each block row, solution speed can be improved by storing this information in memory. Given n states, this can be done at an average cost of a vector of n bytes per node.

Observations show that, for very large models where disk caching is ineffective, the performance of the steady-state solver is limited by disk throughput of about 4–6 MB/sec. As a possible faster alternative, experiments were conducted to determine the feasibility of storing the transition matrix in *memory* in compressed form, and then decompressing it on each iteration. This approach is similar to an "on-the-fly" method (c.f. Section 3.2.4), except that the transition matrix is encoded using standard compression algorithms instead of the model description. We used our own memory-based LZ77 compressor [Sal98] and `zlib` compression library routines to store the transition matrix. However, the compression ratios achieved were unremarkable (less than 2:1) and, as for on-the-fly methods, the high CPU overhead during decompression resulted in poor effective data throughputs of under 1.5MB/s.

Other solution methods

Besides the Jacobi and CGS algorithms of Fig. 5.1 and Fig. 5.2, experiments were also conducted with a variety of other numerical methods including:

- **Alternative Krylov subspace methods**, for example BiCGSTAB [Vor92] and BiCGSTAB2 [Gut93b]. Like CGS, these methods are based on matrix-vector multiplication operations and vector-vector inner products so they can be easily parallelised. These methods showed slightly less erratic convergence behaviour than CGS and similar speedups, but at the cost of a higher memory requirement.

- **Methods based on asynchronous iterations [FS99]**, which avoid processor idle time by eliminating as many synchronization points as possible. Processors do not wait for each other, which means they can get out of phase and perform a different number of iterations, often using out-of-date vector elements. Our observations with an asynchronous parallel Gauss-Seidel algorithm confirm the observation in [FS99] that the method generally requires more computation than synchronous methods and "it is only when the load is not well balanced, or when communication between processors is slow that this approach is advantageous."

- **Block two-stage methods [MPS96]**, which are block iterative methods that solve a linear system of form

$$A_{ii}x_i = b_i \qquad 1 \leq i \leq N$$

where A_{ii} are diagonal matrix blocks. When the diagonal blocks are large, their solution is usually approximated using an iterative method. These methods are called two-stage iterative methods and are characterised by outer iterations performed using off-diagonal blocks and inner iterations performed by repeatedly solving diagonal blocks. The Block Gauss-Seidel algorithm used by the on-the-fly method of Section 3.2.4 and the sequential disk-based method of Section 3.2.5 is an example of a two-stage iterative method. Experiments with a parallel two-stage method that used Jacobi for outer iterations and Gauss-Seidel for inner iterations produced poor convergence results on our test models. However, these methods are worthy of further investigation since the reuse of diagonal matrix blocks should result in higher throughput due to better disk caching.

6.2.5 The Parallel Performance Analyser

The final stage of the performance analysis sequence combines the low-level steady-state distribution results with the state space to form more meaningful higher-level performance measures such as throughput or mean buffer occupancy. The parallel performance analyser supports two types of performance measures, viz. *state* and *count* measures. The concept of these two types of measures originated in the HIT-tool [BS87].

A state measure can be used to find the mean, variance and probability distribution of any real function of a state of the system. Examples include the average number of tokens on a place in a Petri net, some transition's enabling probability or the distribution of customers at a particular set of queues in a queueing network. Suppose we are interested in some performance measure m which has value v_i in state i. Given n states, the steady-state distribution $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ and the vector of real expression values $v = (v_1, v_2, \ldots, v_n)$ where v_i is a function of the elements of the ith state descriptor, the mean of m is calculated as:

$$\mathrm{E}[m] = \sum_{i=1}^{n} \pi_i v_i$$

We can regard v_i as the contribution of the ith state towards the value of the state measure. For example, if we are interested in the mean number of tokens on a place p in a Petri net, v_i represents the number of tokens on place p in state i.

The second moment of a state measure m is given by:

$$\mathrm{E}[m^2] = \sum_{i=1}^{n} \pi_i v_i^2$$

and hence the variance of m by:

$$\begin{aligned} \mathrm{Var}[m] &= \mathrm{E}[m^2] - (\mathrm{E}[m])^2 \\ &= \sum_{i=1}^{n} \pi_i v_i^2 - \left(\sum_{i=1}^{n} \pi_i v_i \right)^2 \end{aligned}$$

A count measure is used to determine the mean rate at which a particular event occurs e.g. the mean rate of transition firing gives transition throughput. Given a system with n states, the steady state distribution $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ and a function r_i which returns the rate at which the event occurs when the system is in state i, the mean of a count measure m is given by:

$$\mathrm{E}[m] = \sum_{i=1}^{n} \pi_i r_i$$

The calculation of the variance of a count measure requires transient analysis, which is not supported.

To assist in the computation of performance measures, the user code contains methods for calculating the values of v_i and r_i for each state. Each process of the parallel performance analyser reads the local steady-state distribution into memory, taking into account the renumbering and reassignment of the state space that took place during matrix transposition. The process then reads the local state space state-by-state, using the methods in the user code to calculate the contribution of the state towards each peformance measure. Finally a master process collates the weighted performance measures across all the processes and outputs the results.

6.3 Interface Language Specification

An interface language for a Markov chain analyser must meet several design criteria relating to power of expression and ease of use. In particular:

- The language should be powerful enough to support the flexible high-level description of Markov models based on a variety of formalisms, including Generalised Stochastic Petri nets, Queueing networks, Queueing Petri nets, Stochastic Process Algebras etc.

- It should be possible to verify basic functional properties which should hold on the model. Facilities should be provided to specify system invariants that apply to every state and to check for the existence of deadlocks.

- It should be possible to specify a variety of performance results, including *state measures* which compute the value of a real expression at every state (such as buffer occupancy) and *count measures* which measure the occurrence rate of events (such as transition throughput).
- The language should use concepts and constructs that are likely to be familiar to target users.

To meet these goals, we have adapted the interface language used by the sequential Markov chain analyser DNAmaca [Kno96]. The language has a simple TEX-like syntax and uses elementary C/C++ expressions to specify model components, system invariants and performance results. Consequently, it should be familiar to users in academic environments while providing sufficient expressive power.

The following symbols are used in the definition:

`{ X }*`	denotes one or more occurrences of X
`|`	separates alternatives

As in TEX, comments begin with `%`; the remainder of the input line is ignored.

6.3.1 Model Description

The underlying Markov chain describing a system may contain many millions of states and transitions. To avoid explicit enumeration of all of these states and transitions, it is necessary to use a high-level model description. The model description specifies the components of a general state of the system, the rules for and effects of transitions between states and an initial state of the system.

The state space generator maps this high-level description onto a low-level representation consisting of the state space and state transition matrix.

```
model_description = \model {
```

```
  {
    state_vector | initial_state | transition_declaration |
    constant | help_value | invariant | state_output_function |
    partitioning_hash_function | primary_hash_function |
    secondary_hash_function | additional_headers
  }*
}
```

State Descriptor Vector

The *state descriptor vector* consists of a finite set of discrete components which, when taken together, describe a state of the system; each unique assignment to these components corresponds to one state.

A vector of elementary C++ variables (`int`, `long`, `short`, `char` etc.) is ideal for this purpose. Elements with `float` or `double` types are also allowed, although user-specified partitioning, primary and secondary hash functions will need to be supplied in this case. Variables are declared in the same manner as they are in C/C++:

```
state_vector = \statevector{
  { <type> <identifier> {, <identifier> }*; }*
}

type = basic C/C++ variable type;
identifier = valid C/C++ identifier;
```

Initial State

An initial state must be specified as the starting point of the reachability analysis; this can be done using simple C/C++ assignments to the elements of the state vector.

```
initial_state = \initialstate {
  { <assignment> }*
}

assignment = C/C++ assignment to elements of the state vector
```

Transition declarations

Transitions describe how the system moves from state to state (via updates to the current state vector). Since it would be virtually impossible to enumerate successor transitions for every individual reachable state, a more general scheme (similar to USENUM [Scz87]) is used. Possible transitions from the current state are specified by describing:

- one or more *enabling conditions* involving elements of the state vector corresponding to the *current state*.
- an *action* to be taken if the transition is executed; this will involve an assignment to the state vector elements of the *next state*.
- an indication of whether the transition from the current to the next state is *timed* or *instantaneous* (i.e. the transition takes no time to execute).
- a *rate* (for timed transitions) or *relative weight* (for instantaneous transitions) must also be specified; note that these rates and weights may be denoted by (possibly state-dependent) arbitrary expressions. If a non-positive rate or weight is encountered during state exploration, the corresponding transition firing will be ignored during analysis.
- an optional *priority* which allows transitions of a higher priority to preempt lower priority transitions of the same type (i.e. timed or instantaneous).

Transitions from the *current* to the *next* state descriptor vector can be achieved through C/C++ assignment statements, while enabling conditions can be given using C/C++ boolean expressions. Since the conditions and actions will form part of the transition code encapsulated in a C++ `State` object, elements of the *current* state descriptor can be referred to directly while elements of the *next* state descriptor can be accessed via a `next` pointer.

```
transition_declaration = \transition{<identifier>}{
  \condition{<boolean expression>}
  \action{ { <assignment> }* }
  \rate{<real expression>} | \weight{<real expression>}
  \priority{<non-negative integer>}
}

boolean expression = C/C++ boolean expression
real expression = C/C++ real expression
assignment = C/C++ assignment
```

Constants and Help Values

It is convenient to allow for *constant* declarations and complicated formulae which are used repeatedly during the evaluation of transition conditions and rates/weights. Such values are called *help values*; this is a concept adopted from the USENUM sequential Markov chain analyser [Scz87].

```
constant = \constant{<identifer>}{value}

help_value = \helpvalue{<type>}{<identifier>}{<expression>}
```

Invariants

Depending on the application domain, there may be invariant conditions which should not be violated during the generation of the state space; these invariant conditions can be expressed as C/C++ boolean expressions. The state generator will issue a warning if it encounters any state which violates an invariant.

```
invariant = \invariant{<boolean expression>}
```

Custom state output function (optional)

If a deadlock or a violation of a user-specified invariant occurs, the state generator reports the event and outputs the state responsible for the error. A simple default output function is provided; however, the user can also provide a custom output function if desired.

```
state_output_function = \output {
  { <statements> }*
}

statements = C++ statements to output elements of the state vector
```

Custom state hash functions (optional)

The state generator uses a probabilistic method of state space storage (c.f. Chapter 4) which requires the computation of three hash keys for each state. The partitioning hash key is an unsigned 32-bit integer which the state generator uses to determine which processor a state should be assigned to (the state generator performs the necessary **mod** operation depending on the number of available processors). The primary hash key is an unsigned integer which determines the hash table row the state should be inserted into (from 0 to 750 018), while the secondary hash key is a 40-bit integer used as a compressed state value.

Default hash functions that aim to provide a random distribution of states over processors, hash table rows and compressed state values are provided (c.f. Section 4.8). However, users may which to use application-specific knowledge to write a set of functions which minimize the probability that two distinct states map onto the same hash keys.

```
partitioning_hash_function = \taskhash {
```

```
  <C++ function body for the function:
     unsigned int task_key()
   which returns a 32-bit hash key>
}

primary_hash_function = \primaryhash {
  <C++ function body for the function
     unsigned int primary_key()
   which returns an integer from 0 to 750018>
}

secondary_hash_function = \secondaryhash {
  <C++ function body for the function
     void secondary_key(unsigned int &sec1, unsigned char &sec2)
  which returns a 40-bit integer by assigning to the 32-bit
  unsigned integer sec1 and the 8-bit character sec2>
}
```

Additional headers

Should the user require any C/C++ functions which are not usually included by default (such as the advanced mathematical functions to be found in `math.h`), the necessary `#include` statements can be placed in a `header` declaration. Class definitions of user-defined classes can also be placed here.

```
additional_headers = \header {
  <C++ include statements and/or class definitions>
}
```

6.3.2 Generation Control

The user is able to control aspects of the state generation process, such as the *maximum number* of states to be generated or the *maximum cpu time* that should be spent on the generation. The user can also specify the level of feedback by specifying the *report style* and the *report interval.*

```
generation_control = \generation {
  { \maxstates{<long int>} | \maxcputime{<seconds>} |
    \reportstyle{full | short | none} | \reportinterval{<long int>}
  }
}
```

6.3.3 Solution Control

Once the state space has been generated (using the model description), the resulting state transition matrix must be solved for its steady state distribution. The user is able to guide this steady-state solution process through parameters such as:

- *Solution Method.* Possible solution methods include the Jacobi and Conjugate Gradient Squared (CGS) methods.
- *Accuracy.* This specifies the convergence criterion for the iterative methods. These methods will terminate after k iterations with an "accuracy" of ϵ if:

  $$\frac{||r^{(k)}||_\infty}{||x^{(k)}||_\infty} < \epsilon$$

  where $r^{(k)}$ is the residual at the kth iteration and ϵ can vary between 10^{-2} and $2.22045 * 10^{-16}$ (IEEE-754 machine epsilon for double precision). Reported performance results are rounded to reflect this accuracy. If the accuracy is not specified, the default is $\epsilon = 10^{-10}$.

- *Maximum Iterations* (within which iterative methods should converge)

As with the generation of the state space, the user is able to set the required level of reporting feedback.

```
solution_control = \solution {
  { \method{jacobi | cgs} |
    \accuracy{<real>} |
    \maxiterations{<long int>} |
    \reportstyle{full | short | none} |
    \reportinterval{<long int>}
  }*
}
```

6.3.4 Performance Measures/Results

Performance results provide a backward mapping from low-level results like probabilities of states and rates of transitions to higher-level quantities like throughput or mean buffer occupancy. Performance measures can generally be classified as *state* or *count* measures (c.f. Section 6.2.5).

```
\performance_measures = \performance {
  { state_measure | count_measure }*
}
```

State measures

A state measure is used to determine the mean and variance of a real expression which is defined at every state in the system, e.g. the average number of tokens on a particular place of a Petri net or some transition's enabling probability. The mean, variance, standard deviation and probability distribution of state measures can be computed.

```
state_measure = \statemeasure{identifier}{
  \estimator{ {mean | variance | stddev | distribution}* }
  \expression{<real_expression>}
}
```

Count measures

A count measure is used to determine the mean rate at which a particular event occurs e.g. the rate at which a transition fires yields transition throughput.

The occurrence of an event is specified according to three conditions:

- a *precondition* on the current state that must be true.
- a *postcondition* on the next state that must be true.
- *transitions* which must be fired during the transition from the current to the next state.

The conditions can be specified as C++ expressions while the transitions can be given in a list. Note that only the mean of count measures is available, since computation of higher moments requires transient analysis.

```
count_measure = \count_measure{identifer}{
  \estimator{mean}
  \precondition{<boolean expression>}
  \postcondition{<boolean expression>}
  \transition{ all | {<identifier>}* }
}
```

6.4 Example System Specification

The section illustrates how the interface language can be used to specify a real system.

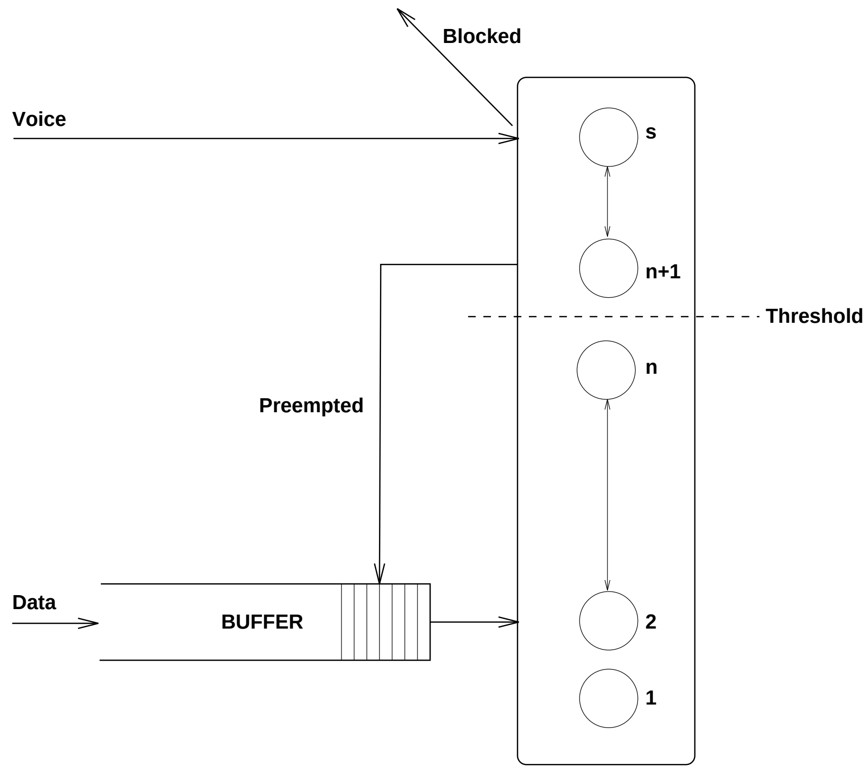


Figure 6.3: A multimedia switch for handling voice and data traffic

Consider the multimedia teletraffic switch designed to handle delay-sensitive voice traffic and delay-insensitive data traffic shown in Fig. 6.3 [AK93, pg. 133–137]. The switch has a capacity for s calls and is designed to give priority to voice calls. If the switch is full and the number of data calls in the system exceeds a certain threshold n, an arriving voice call may preempt a data call. If there are less than n data calls and no free circuits in the switch, arriving voice calls will be blocked. Waiting or preempted data calls are stored in a buffer with capacity b.

There are v potential sources of voice calls. Each of these sources is governed by a two-state Markov process which alternates between a silence phase and a talkspurt phase. The mean duration of the silence phase is $1/\lambda_1$ and the mean duration of a talkspurt phase is $1/\mu_1$. The data call arrival process is simpler, being Poisson with parameter λ_2. Data calls are served at a rate of μ_2 per circuit.

The interface language specification for this system is given below. We assume that the switch capacity $s = 72$, the preemption threshold $n = 50$, the buffer size

$b = 200$ and that there are $v = 1000$ voice sources, with $\lambda_1 = 0.04$ and $\mu_1 = 1.0$. The data arrival rate is $\lambda_2 = 43.0$, and the data service rate is $\mu_2 = 1.2$ per circuit.

```
\model{
  \constant{ss}{72}              % servers in switch
  \constant{voice_source}{1000} % voice sources
  \constant{buffer_size}{200}   % buffer size
  \constant{threshold}{50}      % preemption threshold
  \constant{lambda_0}{0.04}     % voice silence 0->1
  \constant{lambda_1}{1.0}      % talk spurt 1->0
  \constant{lambda_2}{43.0}     % data call arrival rate
  \constant{mu_2}{1.2}          % data service rate

  \statevector{
    \type{int}{data,voice,buffer}
  }

  \helpvalue{int}{idle_voice_source}{voice_source - voice}
  \invariant{ (voice + data) <= ss }
  \initial{
    data = 0;
    voice = 0;
    buffer = 0;
  }

  \transition{data_arrival}{
    \condition{buffer < buffer_size}
    \action{ next->buffer = buffer + 1; }
    \rate{lambda_2}
  }

  \transition{serve}{
    \condition{buffer > 0 && voice + data < ss}
    \action{
      next->buffer = buffer - 1;
      next->data = data + 1;
    }
    \weight{1.0}
  }

  \transition{data_service}{
    \condition{data > 0}
    \action{ next->data = data - 1; }
    \rate{(double)mu_2*data}
```

```
  }

  \transition{voice_arrival}{
    \condition{voice < ss && idle_voice_source}
    \action{
      if ( ((voice + data) >= ss) && (data > threshold)) {
if (buffer < buffer_size)
  next->buffer = buffer + 1;
        next->data = data - 1;
        next->voice = voice + 1;
      } else if ( ((voice + data) >= ss) && (data <= threshold) ) {
/* cannot preempt --> discard call */
      } else if ((voice + data) < ss) {
next->voice = voice + 1;
      }
    }
    \rate{ (double) lambda_0*idle_voice_source}
  }

  \transition{voice_service}{
    \condition{voice > 0}
    \action{ next->voice = voice - 1; }
    \rate{ (double) lambda_1*voice}
  }

}

\performance{
  \statemeasure{mean voice} {
    \estimator{mean variance distribution}
    \expression{voice}
  }

  \statemeasure{mean data} {
    \estimator{mean variance distribution}
    \expression{data}
  }

  \statemeasure{mean buffer} {
    \estimator{mean variance distribution}
    \expression{buffer}
  }

  \countmeasure{blocking rate} {
    \estimator{mean}
    \precondition{1}
```

```
    \postcondition{voice == next->voice}
    \transition{voice_arrival}
  }

  \countmeasure{voice throughput}{
    \estimator{mean}
    \transition{voice_service}
  }

  \countmeasure{data throughput}{
    \estimator{mean}
    \transition{data_service}
  }
}

\solution{
  \method{cgs}
  \accuracy{1e-10}
}
```

Chapter 7

Conclusion

7.1 Summary of Thesis Achievements

This thesis has addressed the challenge of constructing and solving very large continuous time Markov chains derived from dynamic specifications of complex concurrent systems.

Conventional methods for Markov chain analysis are limited to small chains because they have very high time and space requirements. This problem has been attacked on two fronts: parallelism has been used to reduce time requirements, while probabilistic and disk-based storage techniques have been used to reduce space requirements. In contrast to many of the contemporary approaches discussed in Chapter 3, the novel generation and solution techniques developed in Chapter 4 and Chapter 5 do not place any restrictions on the type of system that can be analysed. Nor are the techniques limited to any particular formalism, being applicable to a wide class of stochastic models that includes Stochastic Petri nets, Queueing networks, Queueing Petri nets and Stochastic Process Algebras.

The parallel dynamic probabilistic state space generation algorithm described in Chapter 4 is the first major contribution of this work. This algorithm rapidly enumerates the large number of low-level states that a system can enter and constructs a graph which describes how the system moves from state to state.

The use of hash compaction means that memory requirements are very low and independent of the length of the state vector. Since the method is probabilistic, there is a risk that states may be misidentified or omitted from the state graph. However, this probability has been quantified and has been shown to be extremely low. Uniquely for a probabilistic scheme, the algorithm makes use of dynamic memory allocation. This avoids the problems of over or under-allocation associated with a traditional static memory allocation. Further, the algorithm parallelises well and has an enhanced variant with a low communication overhead. A theoretical performance model shows that the algorithm delivers good speedups and exhibits good scalability. These properties are confirmed by results from an implementation running on a Fujitsu AP3000 distributed memory parallel computer. State spaces of over 100 million states and 1 billion arcs are generated across 16 nodes in just over half an hour, with an omission probability of under 0.05%, i.e. there is a 99.95% chance that the state graph has been generated in its entirety without omitting or misidentifying any states whatsoever. Further, the omission probability may be arbitrarily reduced at logarithmic runtime cost.

The distributed disk-based solution technique of Chapter 5 is the second major contribution of this work. This technique determines the long-run probability of being in each state by mapping the state graph onto a Markov chain and then solving a large sparse set of linear equations derived from the chain. The focus is on two numerical methods that are suited to parallel implementation, viz. the Jacobi and Conjugate Gradient Squared (CGS) algorithms. Parallel sparse matrix-vector multiplication emerges as the critical bottleneck in these methods. The efficiency of this operation is improved in two ways. Firstly, the states of the chain's transition matrix are remapped in order to improve data locality. Secondly, the remapped matrix is distributed across processors such that the number of non-zero elements allocated to each node is the same, thus ensuring a good load balance. Per processor memory requirements are low since transition matrix elements are not stored in memory but are kept on disk and read in as needed. In addition, all vectors held in memory are completely distributed and the number of vectors stored in memory is kept to a minimum by writing unused intermediate vectors to disk. We have described and implemented a distributed high performance solver architecture that makes use of two processes per node to overlap disk I/O with computation and communication. The resulting parallel solver delivers good speedups and is able to solve systems of the order of 100 million states and 1 billion transitions. Solving such a large problem on a single machine would be a daunting task indeed – besides the huge amount of computation required, the memory required to store the solution vector alone would be over 800MB.

To place the main contributions in context, the remaining tools for a complete parallel performance analysis pipeline have been implemented in C++ using the Message Passing Interface (MPI) standard. Chapter 6 describes the final toolset which comprises an interface language parser, a parallel state space generator, a parallel matrix transposer, a parallel steady-state solver and a parallel performance analyser. This pipeline provides an efficient and seamless way of automatically obtaining performance measures for unrestricted high-level system specifications with very large underlying state spaces and state graphs.

7.2 Applications

In this section, we highlight the applicability of our contributions to the general area of the correctness and performance analysis of concurrent systems.

Our perspective in this thesis has been focused on the parallel generation and solution of large Markov models *for the purposes of performance analysis*. However, the ability to generate and manipulate large state spaces derived from model specifications is also of interest to those concerned with the *correctness analysis* of concurrent systems. In this context, our parallel state space generation algorithm can be applied to the construction of the labelled transition systems used by researchers in the model checking community. Support for this can be provided by simply annotating each arc in the state graph with an action name rather than a transition rate. Furthermore, the existing state space generation algorithm already provides basic correctness checks, such as deadlock detection and detection of "unsafe" states through the assertion of user-specified invariants. Although our parallel solution algorithm has less relevance to correctness analysis, model checking algorithms also manipulate very large transition matrices. Consequently, a parallel disk-based approach may be appropriate when verifying certain logical formulae on very large state spaces.

Although we have considered unrestricted, non-hierarchical systems, our techniques can also be applied to structured methods for compositional reachability analysis [Gia99] and compositional performance analysis [Hil94, Her99]. These methods generate and solve submodels in a modular fashion, and apply state space reduction rules as submodels are synthesised. Support for this can be provided by extending the interface language to allow for the specification of submodels and by extending the state generator so that it can combine independently generated submodel state spaces subject to synchronisation constraints. The resulting framework would support larger, less restricted submodels, and would allow for very large intermediate and final state space sizes.

Finally, while our framework incorporates explicit support only for transitions that fire immediately or after an exponentially distributed delay, general (non-exponential) firing delays can be approximated by using phase-type distributions. A phase-type distribution is the distribution of the time to absorption in a network of exponential stages. Each exponential stage i is associated with a rate μ_i and a set of routing probabilities p_{ij} describing the probabilities of routing a client from stage i to stage j (where j includes the absorbing stage). A good example of a phase-type distribution is the Erlang$_n$ distribution, which is formed as a series of n exponential stages with the same rate μ. This distribution is often used to approximate a deterministic delay of length n/μ. Since phase-type distributions can themselves be specified at a low-level as CTMCs, no modifications to the interface language are required to support them, although extensions allowing for their compact specification could be added. Note that the price paid for this increased modelling realism is an increased state space size (since now the stage of the each phase-type distribution is an additional component of the state descriptor).

7.3 Future Work

This section discusses some ideas for future work that would improve the capacity, efficiency and applicability of our methods.

The capacity of the parallel state space generation algorithm could be improved by using magnetic disk to support a larger state hash table. Since the state hash table is accessed randomly, this does not at first appear to be feasible. However, lookup operations on states in the state table can be delayed, and then periodically checked *en masse* against a state table that is read from disk in a linear fashion. This idea has already been shown to work well using a static probabilistic algorithm in a single processor context [SD98], where storing over 95% of the state table on disk resulted in a slowdown of only 25%. There is no reason to suspect that equally good results could not be obtained in a parallel context using our dynamic algorithm.

The capacity and efficiency of the parallel disk-based solver could be improved by developing scalable numerical methods that have lower memory requirements and lower communication overheads, and that reuse matrix blocks as they are generated. Lower communication overheads and higher processor utilizations could be achieved by using asynchronous iterations [FS99]. Under this approach all synchronisation points between processors are eliminated and processors exchange data on an infrequent, asynchronous basis. Although in general more computation is required to achieve convergence relative to a synchronous scheme, communication overhead is lower and all idle time is eliminated. Reuse of matrix blocks could be accomplished using two stage methods [MPS96]. During each "outer" iteration, these methods reuse diagonal matrix blocks several times to perform several "inner" iterations. However, in order to avoid load balancing problems, the distribution of states over processors would need to be adjusted according to the size of the diagonal blocks.

It is possible to solve for the stationary distribution of a finite continuous-time Markov chain if and only if the chain is irreducible, i.e. if every state communicates with every other state. Therefore, a useful addition to our toolset would be a parallel functional analyser that performs a strongly connected component analysis of the states in a given state graph. If transient states are found, they should be eliminated, and if more than one strongly connected component exists, the states in each component should be analysed as separate Markov chains. Unfortunately, all known efficient algorithms for the connected component analysis of directed graphs require some sort of depth-first search of the state graph, which makes them difficult to parallelise [Ste97]. Consequently it would be interesting to investigate connected component analysis algorithms that are based on a breadth-first search strategy and thus amenable to parallel implementation.

In real systems, performance targets are often specified in terms of response time distributions. For example, in a transaction processing environment it might be required that 95% of all transactions complete in under 30 seconds. In such situations, it is useful to know the distribution of the passage time [She93, Kul95] between two given states in the state graph. Convolving the distributions of the state holding times across all possible paths between the states is a computationally expensive task, but one that is well suited to parallel implementation.

Finally, although transitions with non-exponential firing delays can be approximated using phase-type distributions, systems with exponential and deterministic firing delays could be modelled and analysed exactly by using the techniques proposed by Lindemann for Deterministic and Stochastic Petri nets (DSPNs) [Lin98]. Steady state analysis of such models that have concurrently enabled deterministic transitions typically involves the solution of several very large systems of integral and linear equations. A parallel disk-based approach could therefore be beneficial.

Appendix A

Model Descriptions

A.1 Introduction

This appendix presents the full specification and description of the two models that were used as demonstration examples in Chapters 4 and 5. Section A.2 describes a model of a flexible manufacturing system, while Section A.3 presents a model of software used in the Courier telecommunications protocol. Both models have a scalable parameter which allows for the generation of state spaces and state graphs of various sizes. Consequently, they have been widely used in the literature as testbeds for state space generation and/or steady state solution algorithms.

A.2 The FMS Model

A.2.1 Model Description

Fig. A.1 shows a 22-place Generalized Stochastic Petri net model of a flexible manufacturing system. This model, which we will refer to as the FMS model, was initially described in [CT93] and was subsequently used in [CGN98] to demonstrate distributed exhaustive state space generation.

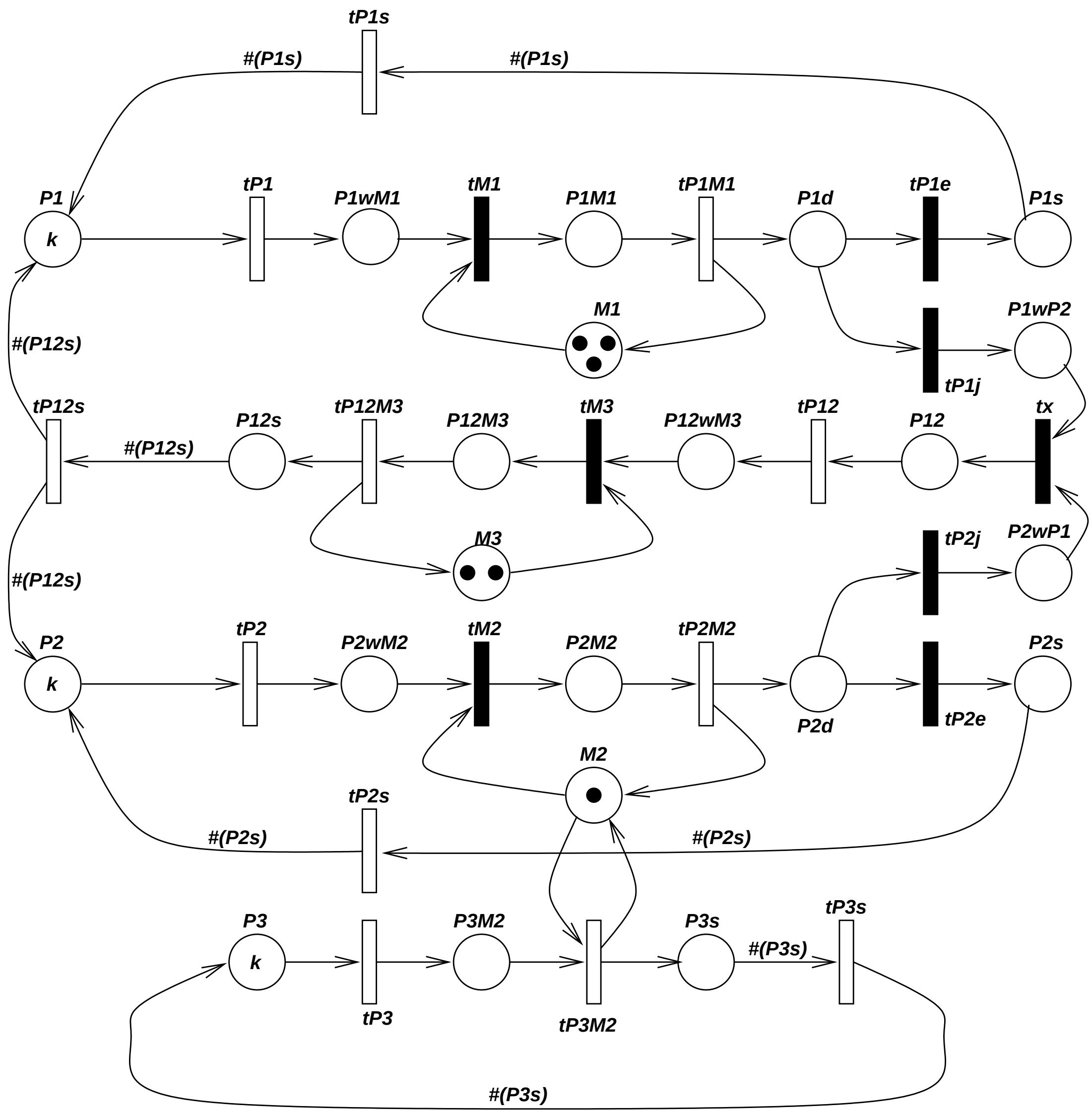


Figure A.1: The FMS Generalised Stochastic Petri net [CT93].

The FMS model describes an assembly line with three types of machines ($M1, M2$ and $M3$) which assemble four types of parts ($P1$, $P2$, $P3$ and $P12$). Initially there are k unprocessed parts of each type $P1, P2$ and $P3$ in the system. There are no parts of type $P12$ at start-up since these are assembled from processed parts of type $P1$ and $P2$. For our purposes, k is a useful scaling parameter since different values of k produce underlying state graphs with different numbers of states and transitions.

Three machines of type $M1$ process parts of type $P1$. One machine of type $M2$ processes parts of type $P2$, although this machine can also process parts of type $P3$ if no parts of type $P2$ are available. Two machines of type $M3$ assemble parts of type $P1$ and $P2$ to form parts of type $P12$. When parts of any type are finished, they can be *shipped* (via transitions $tP1s, tP2s, tP3s, tP12s$), in which case the same number of unprocessed parts enters the system to maintain a constant inventory.

Unprocessed parts of types $P1, P2$ and $P3$ are moved between machines using pallets, each of which holds one part. Processed parts of type $P1$ and $P2$ (required by $M3$ to make parts of type $P12$) also need to be moved in this way, but two (one of each type) can share one pallet. The total number of pallets *available* in the system remains constant and is given by $N_p = \lfloor 3k/2 \rfloor$, while the total number of pallets *requested* in a given marking is $r = \#(P1) + \#(P2) + \#(P3) + \#(P12)$. If the number of pallets requested exceeds the total number of pallets available, contention for the pallets is approximated using a processor-sharing policy.

Table A.1 presents the rates and weights, many of them state-dependent, assigned to the transitions in the FMS model. Note how the sharing of the pallets (in terms of the number available N_p and the number required r) is reflected in the rates assigned to transitions $tP1, tP2, tP3$ and $tP12$. The immediate transitions $tP1e$ and $tP1j$ (resp. $tP2e$ and $tP2j$) describe the relative likelihood that a finished part of type P1 (resp. P2) will *exit* the system (and be shipped) as opposed to *joining* with a part of type P2 (resp. P1) to form a part of type P12.

Timed Transition	Rate ($\min^{-1}$)
$tP1$	$\#(P1)\ \min(1,\ N_p/r)$
$tP2$	$\#(P2)\ \min(1,\ N_p/r)$
$tP3$	$\#(P3)\ \min(1,\ N_p/r)$
$tP12$	$\#(P12)\ \min(1,\ N_p/r)$
$tP1M1$	$\#(P1M1)/4$
$tP2M2$	$1/6$
$tP3M2$	$1/2$
$tP12M3$	$\#(P12M3)$
$tP1s$	$1/60$
$tP2s$	$1/60$
$tP3s$	$1/60$
$tP12s$	$1/60$
Immediate Transition	**Weight**
$tP1e$ vs. $tP1j$	0.8 vs. 0.2
$tP2e$ vs. $tP2j$	0.6 vs. 0.4

Table A.1: Transition rates and weights in the FMS model.

A.2.2 Performance Measures

Ciardo and Trivedi propose a single performance measure ψ describing the "productivity" of the FMS:

$$\psi = 400\phi_1 + 600\phi_2 + 100\phi_3 + 1100\phi_{12}$$

where ϕ_x, $x \in \{1, 2, 3, 12\}$ is the throughput (per minute) for parts of type x (given by the throughput of transitions $tP1, tP2, tP3$ and $tP12$ respectively). The multiplicative constants reflect the net benefit of producing a part of the corresponding type.

Table A.2 presents the calculated values of ψ for different values of k, as published in [CT93] (for $k \leq 5$), and as calculated by the parallel performance analyser (for $k \leq 11$). The computed results for $k \leq 5$ agree with the published results to 3 significant figures. The low accuracy used to compute the published results probably accounts for the lack of agreement observed beyond this precision.

k	ψ
1	13.853148
2	29.154731
3	44.443713
4	59.551361
5	74.373573

k	ψ
1	13.867015
2	29.154690
3	44.444877
4	59.551291
5	74.374123
6	88.851914
7	102.944053
8	116.624587
9	129.876031
10	142.686488
11	155.046937

Table A.2: Published (left) and computed (right) values of the FMS "productivity" performance measure ψ, in terms of k, the initial number of unprocessed parts of type $P1, P2$ and $P3$.

A.2.3 Analyser Input File

The FMS model can be specified using the interface language of Section 6.3 as follows:

```
\model{

  % FMS flexible manufacturing system from Ciardo and Trivedi 1993

  \constant{kk}{9}

  \statevector{
    \type{short}{P1, P1wM1, P1M1, P1d, P1s, M1, P1wP2}
    \type{short}{P12s, P12M3, M3, P12wM3, P12}
    \type{short}{P2, P2wM2, P2M2, M2, P2d, P2s, P2wP1}
    \type{short}{P3, P3M2}
    \type{short}{P3s}
  }

  \initial{
    P1 = kk;
    P1wM1 = 0;
    P1M1 = 0;
    P1d = 0;
    P1s = 0;
    M1 = 3;
    P1wP2 = 0;
    P12s = 0;
```

```
  P12M3 = 0;
  M3 = 2;
  P12wM3 = 0;
  P12 = 0;
  P2 = kk;
  P2wM2 = 0;
  P2M2 = 0;
  M2 = 1;
  P2d = 0;
  P2s = 0;
  P2wP1 = 0;
  P3 = kk;
  P3M2 = 0;
  P3s = 0;
}

\constant{Np}{3*kk/2}
\helpvalue{int}{r}{(P1+P2+P3+P12)}
\helpvalue{double}{min}{ (Np < r) ? (double) Np/r : 1.0 }

\transition{tP1s}{
  \condition{P1s > 0}
  \action{next->P1s = 0; next->P1 += P1s; }
  \rate{1.0/60.0}
}

\transition{tP1}{
  \condition{P1 > 0}
  \action{next->P1 = P1 - 1; next->P1wM1 = P1wM1 + 1; }
  \rate{(double) P1*min}
}

\transition{tM1}{
  \condition{(P1wM1 > 0) && (M1 > 0)}
  \action{
    next->P1wM1 = P1wM1 - 1; next->M1 = M1 - 1;
    next->P1M1 = P1M1 + 1;
  }
  \weight{1.0}
}

\transition{tP1M1}{
  \condition{P1M1 > 0}
  \action{
    next->P1M1 = P1M1 - 1;
    next->M1 = M1 + 1; next->P1d = P1d + 1;
```

```
  }
  \rate{(double) P1M1/4.0}
}

\transition{tP1e}{
  \condition{P1d > 0}
  \action{next->P1d = P1d - 1; next->P1s = next->P1s + 1; }
  \weight{0.8}
}

\transition{tP1j}{
  \condition{P1d > 0}
  \action{next->P1d = P1d - 1; next->P1wP2 = next->P1wP2 + 1; }
  \weight{0.2}
}

\transition{tP12s}{
  \condition{P12s > 0}
  \action{
    next->P12s = 0;
    next->P1 = P1 + P12s; next->P2 = P2 + P12s;
  }
  \rate{1.0/60.0}
}

\transition{tP12M3}{
  \condition{P12M3 > 0}
  \action{
    next->P12M3 = P12M3 - 1; next->P12s = P12s + 1;
    next->M3 = M3 + 1;
  }
  \rate{(double) P12M3}
}

\transition{tM3}{
  \condition{(M3 > 0) && (P12wM3 > 0)}
  \action{
    next->P12wM3 = P12wM3 - 1; next->M3 = M3 - 1;
    next->P12M3 = P12M3 + 1;
  }
  \weight{1.0}
}

\transition{tP12}{
  \condition{P12 > 0}
  \action{next->P12 = P12 - 1; next->P12wM3 = P12wM3 + 1; }
```

```
  \rate{(double) P12*min}
}

\transition{tx}{
  \condition{(P1wP2 > 0) && (P2wP1 > 0)}
  \action{
    next->P1wP2 = P1wP2 - 1; next->P2wP1 = P2wP1 - 1;
    next->P12 = P12 + 1;
  }
  \weight{1.0}
}

\transition{tP2}{
  \condition{P2 > 0}
  \action{next->P2 = P2 - 1; next->P2wM2 = P2wM2 + 1; }
  \rate{(double) P2*min}
}

\transition{tM2}{
  \condition{(P2wM2 > 0) && (M2 > 0)}
  \action{
    next->P2wM2 = P2wM2 - 1; next->M2 = M2 - 1;
    next->P2M2 = P2M2 + 1;
  }
  \weight{1.0}
}

\transition{tP2M2}{
  \condition{P2M2 > 0}
  \action{
    next->P2M2 = P2M2 - 1;
    next->P2d = P2d + 1; next->M2 = M2 + 1;
  }
  \rate{1.0/6.0}
}

\transition{tP2j}{
  \condition{P2d > 0}
  \action{ next->P2d = P2d - 1; next->P2wP1 = P2wP1 + 1; }
  \weight{0.4}
}

\transition{tP2e}{
  \condition{P2d > 0}
  \action{ next->P2d = P2d - 1; next->P2s = P2s + 1; }
  \weight{0.6}
```

```
  }

  \transition{tP2s}{
    \condition{P2s > 0}
    \action{next->P2s = 0; next->P2 = P2 + P2s; }
    \rate{1.0/60.0}
  }

  \transition{tP3}{
    \condition{P3 > 0}
    \action{next->P3 = P3 - 1; next->P3M2 = P3M2 + 1;}
    \rate{(double) P3*min;}
  }

  \transition{tP3M2}{
    \condition{(M2 > 0) && (P3M2 > 0)}
    \action{next->P3M2 = P3M2 - 1; next->P3s = P3s + 1;}
    \rate{1.0/2.0}
  }

  \transition{tP3s}{
    \condition{P3s > 0}
    \action{next->P3s = 0; next->P3 = P3 + P3s; }
    \rate{1.0/60.0}
  }
}

\performance{
  \statemeasure{idle machines of type M1}{
    \estimator{mean variance distribution}
    \expression{M1}
  }

  \statemeasure{idle machines of type M2}{
    \estimator{mean variance distribution}
    \expression{M2}
  }

  \statemeasure{idle machines of type M3}{
    \estimator{mean variance distribution}
    \expression{M3}
  }

  \countmeasure{throughput of parts of type P1}{
    \estimator{mean}
    \transition{tP1}
```

```
  }

  \countmeasure{throughput of parts of type P2}{
    \estimator{mean}
    \transition{tP2}
  }

  \countmeasure{throughput of parts of type P3}{
    \estimator{mean}
    \transition{tP3}
  }

  \countmeasure{throughput of parts of type P12}{
    \estimator{mean}
    \transition{tP12}
  }
}

\solution{
  \method{cgs}
}
```

A.3 The Courier Protocol Model

A.3.1 Model Description

Fig. A.2 shows a 45-place Generalized Stochastic Petri net model of software used in the Courier telecommunications protocol. This model was initially described in [WL91] and was subsequently used in [DS97] and [DS98b] to demonstrate sequential disk-based solution techniques.

The Courier Petri net models the ISO Application, Session and Transport network layers involved in the flow of data from a sender ($p1$ to $p26$) to a receiver ($p27$ to $p46$) via a network (c.f. Fig. A.2). Messages are conveyed between layers by "Courier" tasks which act as active data buffers.

The transport layer fragments data, which is modelled as two paths between $p13$ and $p33$. The path via $t8$ carries all fragments before the last one to $p33$. Acknowledgements for these fragments are sent back to the sender, but no data

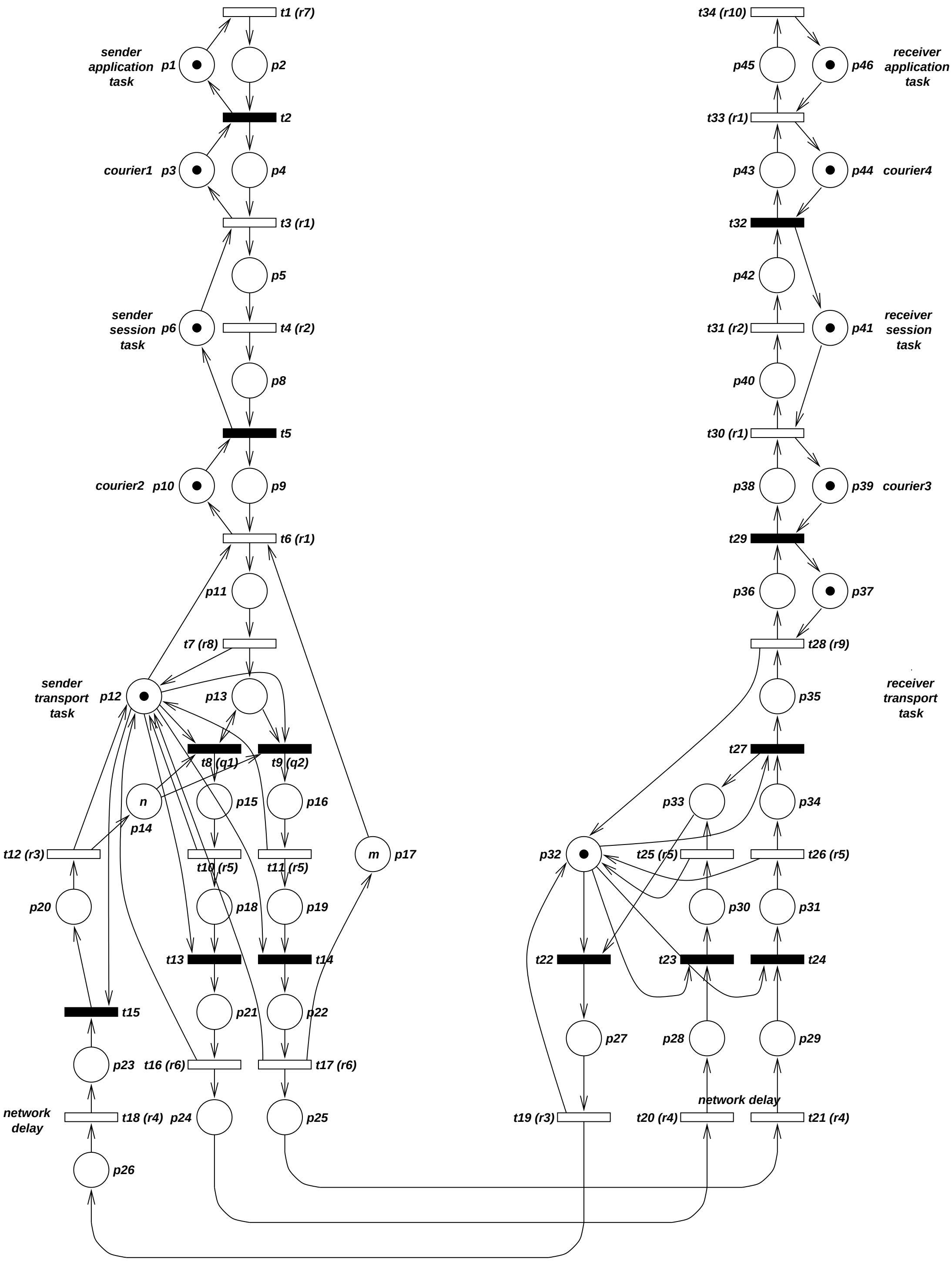


Figure A.2: The Courier Protocol Software Generalised Stochastic Petri net [WL91].

is delivered to the higher layers on the receiver side. The path via $t9$ carries the last fragment of each message block. Acknowledgements for these fragments are generated and a data token is delivered upwards via $t27$. The average message length is determined by the ratio of the weights on the immediate transitions $t8$ and $t9$. This ratio, known as the fragmentation ratio, is given by $q1 : q2$ (where $q1$ and $q2$ are the weights associated with transitions $t8$ and $t9$ respectively). As in [DS97], we will assume a fragmentation ratio of one.

The transport layer is further characterized by two important parameters: the sliding window size k ($p14$) and the transport space m ($p17$). As in [DS97], we will set $m = 1$ and vary k to produce underlying state graphs of different sizes.

The rates of the timed transitions in the model were determined by observing an implementation of the software running on a Sun workstation. 5000 messages of 1000-bytes each were sent and the execution time noted. Table A.3 presents the resulting transition rates and weights.

Timed Transition Rate	Value
$r1$	5000/0.57
$r2$	5000/4.97
$r3$	5000/1.09
$r4$	5000/10.37
$r5$	5000/4.29
$r6$	5000/0.39
$r7$	5000/0.68
$r8$	5000/2.88
$r9$	5000/3.45
$r10$	5000/1.25
Immediate Transition Weight	Value
$q1$ vs. $q2$	1.0 vs. 1.0

Table A.3: Transition rates and weights in the Courier Protocol model.

A.3.2 Performance Measures

Woodside and Li propose several performance measures. The most important is λ, the data throughput rate, which is given by the throughput of transition $t21$.

Further, there are some measures which determine task utilizations. In particular, $P_{transp1} = Pr\{p12 \text{ is marked}\} = Pr\{\text{transport task 1 is idle}\}$. Similarly they define $P_{transp2}$ for $p32$, P_{sess1} and P_{sess2} using $p6$ and $p41$, and P_{send} and P_{recv} using $p1$ and $p46$.

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
λ	74.3467	120.372	150.794	172.011	187.413	198.919
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80197
$P_{transp1}$	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632
$P_{transp2}$	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365

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

Table A.4: Published (top) and computed (bottom) values of the Courier Protocol performance measures in terms of the transport window size k.

Table A.4 presents the calculated values of these performance measures for different values of k, as published in [WL91] (for values of $k \leq 2$) and [DS97] (for values of $3 \leq k \leq 6$). Also presented are the performance measures as calculated by the parallel performance analyser (for values up to $k \leq 8$). The computed results for $k \leq 6$ are in complete agreement with the published results, except for P_{sess2} in the $k = 6$ case (a calculated value of 0.80197 vs. a published value of 0.80196).

A.3.3 Analyser Input File

The FMS model can be specified using the interface language of Section 6.3 as follows:

```
\model{

  % Courier Protocol Model from Woodside and Li 1991

  \constant{kk}{8}
  \constant{mm}{1}

  \constant{r1}{(5000.0/0.57)}
  \constant{r2}{(5000.0/4.97)}
  \constant{r3}{(5000.0/1.09)}
  \constant{r4}{(5000.0/10.37)}
  \constant{r5}{(5000.0/4.29)}
  \constant{r6}{(5000.0/0.39)}
  \constant{r7}{(5000.0/0.68)}
  \constant{r8}{(5000.0/2.88)}
  \constant{r9}{(5000.0/3.45)}
  \constant{r10}{(5000.0/1.25)}

  \constant{q1}{1.0}
  \constant{q2}{1.0}

  \statevector{
    \type{short}{  p1,  p2,  p3,  p4,  p5,  p6,  p8,  p9, p10 }
    \type{short}{ p11, p12, p13, p14, p15, p16, p17, p18, p19 }
    \type{short}{ p20, p21, p22, p23, p24, p25, p26, p27, p28 }
    \type{short}{ p29, p30, p31, p32, p33, p34, p35, p36, p37 }
    \type{short}{ p38, p39, p40, p41, p42, p43, p44, p45, p46 }
  }

  \initial{
     p1 =  p3 =  p6 = p10 = p12 = p32 = 1;
    p37 = p39 = p41 = p44 = p46 = 1;
     p2 =  p4 =  p5 =  p8 = p9  = p11 = 0;
    p13 = p15 = p16 = p18 = p19 = p20 = 0;
    p21 = p22 = p23 = p24 = p25 = p26 = 0;
    p27 = p28 = p29 = p30 = p31 = p33 = 0;
   p34 = p35 = p36 = p38 = p40 = p42 = 0;
    p43 = p45 = 0;
    p14 = kk;
    p17 = mm;
  }

  \transition{t1}{
    \condition{p1 >  0}
    \action{next->p1 = p1 - 1; next->p2 = p2 + 1;}
    \rate{r7}
```

```
}

\transition{t2}{
  \condition{p2 > 0 && p3 > 0}
  \action{
    next->p2 = p2 - 1; next->p3 = p3 - 1;
    next->p1 = p1 + 1; next->p4 = p4 + 1;
  }
  \weight{1.0}
}

\transition{t3}{
  \condition{p4 > 0 && p6 > 0}
  \action{
    next->p4 = p4 - 1; next->p6 = p6 - 1;
    next->p3 = p3 + 1; next->p5 = p5 + 1;
  }
  \rate{r1}
}

\transition{t4}{
  \condition{p5 > 0}
  \action{next->p5 = p5 - 1; next->p8 = p8 + 1;}
  \rate{r2}
}

\transition{t5}{
  \condition{p8 > 0 && p10 > 0}
  \action{
    next->p8 = p8 - 1; next->p10 = p10 - 1;
    next->p6 = p6 + 1; next->p9 = p9 + 1;
  }
  \weight{1.0}
}

\transition{t6}{
  \condition{p9 > 0 && p12 > 0 && p17 > 0}
  \action{
    next->p9 = p9 - 1; next->p12 = p12 - 1; next->p17 = p17 - 1;
    next->p10 = p10 + 1; next->p11 = p11 + 1;
  }
  \rate{r1}
}

\transition{t7}{
  \condition{p11 > 0}
```

```
    \action{
      next->p11 = p11 - 1;
      next->p12 = p12 + 1; next->p13 = p13 + 1;
    }
    \rate{r8}
  }

  \transition{t8}{
    \condition{p12 > 0 && p13 > 0 && p14 > 0}
    \action{
      next->p12 = p12 - 1; next->p14 = p14 - 1;
      next->p15 = p15 + 1;
    }
    \weight{q1}
  }

  \transition{t9}{
    \condition{p12 > 0 && p13 > 0 && p14 > 0}
    \action{
      next->p12 = p12 - 1; next->p13 = p13 - 1; next->p14 = p14 - 1;
      next->p16 = p16 + 1;
    }
    \weight{q2}
  }

  \transition{t10}{
    \condition{p15 > 0}
    \action{
      next->p15 = p15 - 1;
      next->p12 = p12 + 1; next->p18 = p18 + 1;
    }
    \rate{r5}
  }

  \transition{t11}{
    \condition{p16 > 0}
    \action{
      next->p16 = p16 - 1;
      next->p12 = p12 + 1; next->p19 = p19 + 1;
    }
    \rate{r5}
  }

  \transition{t12}{
    \condition{p20 > 0}
    \action{
```

```
    next->p20 = p20 - 1;
    next->p14 = p14 + 1; next->p12 = p12 + 1;
  }
  \rate{r3}
}

\transition{t13}{
  \condition{p12 > 0 && p18 > 0}
  \action{
    next->p12 = p12 - 1; next->p18 = p18 - 1;
    next->p21 = p21 + 1;
  }
  \weight{1.0}
}

\transition{t14}{
  \condition{p12 > 0 && p19 > 0}
  \action{
    next->p12 = p12 - 1; next->p19 = p19 - 1;
    next->p22 = p22 + 1;
  }
  \weight{1.0}
}

\transition{t15}{
  \condition{p12 > 0 && p23 > 0}
  \action{
    next->p12 = p12 - 1; next->p23 = p23 - 1;
    next->p20 = p20 + 1;
  }
  \weight{1.0}
}

\transition{t16}{
  \condition{p21 > 0}
  \action{
    next->p21 = p21 - 1;
    next->p12 = p12 + 1; next->p24 = p24 + 1;
  }
  \rate{r6}
}

\transition{t17}{
  \condition{p22 > 0}
  \action{
    next->p22 = p22 - 1; next->p12 = p12 + 1;
```

```
    next->p17 = p17 + 1; next->p25 = p25 + 1;
  }
  \rate{r6}
}

\transition{t18}{
  \condition{p26 > 0}
  \action{next->p26 = p26 - 1; next->p23 = p23 + 1;}
  \rate{r4}
}

\transition{t19}{
  \condition{p27 > 0}
  \action{
    next->p27 = p27 - 1;
    next->p32 = p32 + 1; next->p26 = p26 + 1;
  }
  \rate{r3}
}

\transition{t20}{
  \condition{p24 > 0}
  \action{next->p24 = p24 - 1; next->p28 = p28 + 1;}
  \rate{r4}
}

\transition{t21}{
  \condition{p25 > 0}
  \action{next->p25 = p25 - 1; next->p29 = p29 + 1;}
  \rate{r4}
}

\transition{t22}{
  \condition{p32 > 0 && p33 > 0}
  \action{
    next->p32 = p32 - 1; next->p33 = p33 - 1;
    next->p27 = p27 + 1;
  }
  \weight{1.0}
}

\transition{t23}{
  \condition{p32 > 0 && p28 > 0}
  \action{
    next->p32 = p32 - 1; next->p28 = p28 - 1;
    next->p30 = p30 + 1;
```

```
  }
  \weight{1.0}
}

\transition{t24}{
  \condition{p32 > 0 && p29 > 0}
  \action{
    next->p32 = p32 - 1; next->p29 = p29 - 1;
    next->p31 = p31 + 1;
  }
  \weight{1.0}
}

\transition{t25}{
  \condition{p30 > 0}
  \action{
    next->p30 = p30 - 1;
    next->p32 = p32 + 1; next->p33 = p33 + 1;
  }
  \rate{r5}
}

\transition{t26}{
  \condition{p31 > 0}
  \action{
    next->p31 = p31 - 1;
    next->p34 = p34 + 1; next->p32 = p32 + 1;
  }
  \rate{r5}
}

\transition{t27}{
  \condition{p34 > 0 && p32 > 0}
  \action{
    next->p34 = p34 - 1; next->p32 = p32 - 1;
    next->p33 = p33 + 1; next->p35 = p35 + 1;
  }
  \weight{1.0}
}

\transition{t28}{
  \condition{p35 > 0 && p37 > 0}
  \action{
    next->p35 = p35 - 1; next->p37 = p37 - 1;
    next->p32 = p32 + 1; next->p36 = p36 + 1;
  }
```

```
    \rate{r9}
  }

  \transition{t29}{
    \condition{p36 > 0 && p39 > 0}
    \action{
      next->p36 = p36 - 1; next->p39 = p39 - 1;
      next->p38 = p38 + 1; next->p37 = p37 + 1;
    }
    \weight{1.0}
  }

  \transition{t30}{
    \condition{p41 > 0 && p38 > 0}
    \action{
      next->p41 = p41 - 1; next->p38 = p38 - 1;
      next->p39 = p39 + 1; next->p40 = p40 + 1;
    }
    \rate{r1}
  }

  \transition{t31}{
    \condition{p40 > 0}
    \action{next->p40 = p40 - 1; next->p42 = p42 + 1;}
   \rate{r2}
  }

  \transition{t32}{
    \condition{p42 > 0 && p44 > 0}
    \action{
      next->p42 = p42 - 1; next->p44 = p44 - 1;
      next->p43 = p43 + 1; next->p41 = p41 + 1;
    }
    \weight{1.0}
  }

  \transition{t33}{
    \condition{p43 > 0 && p46 > 0}
    \action{
      next->p43 = p43 - 1; next->p46 = p46 - 1;
      next->p44 = p44 + 1; next->p45 = p45 + 1;
    }
    \rate{r1}
  }

  \transition{t34}{
```

```
    \condition{p45 > 0}
    \action{next->p45 = p45 - 1; next->p46 = p46 + 1;}
    \rate{r10}
  }
}

\performance {
  \countmeasure{lambda (data throughput)}{
    \estimator{mean}
    \transition{t21}
  }

  \statemeasure{psend}{
    \estimator{mean variance distribution}
    \expression{p1}
  }

  \statemeasure{precv}{
    \estimator{mean variance distribution}
    \expression{p46}
  }

  \statemeasure{psess1}{
    \estimator{mean variance distribution}
    \expression{p6}
  }

  \statemeasure{psess2}{
    \estimator{mean variance distribution}
    \expression{p41}
  }

  \statemeasure{ptransp1}{
    \estimator{mean variance distribution}
    \expression{p12}
  }

  \statemeasure{ptransp2}{
    \estimator{mean variance distribution}
    \expression{p32}
  }
}

\solution{
  \method{cgs}
}
```

Appendix B

AP3000 Technical Overview

B.1 Introduction

This appendix describes the architecture of the Fujitsu AP3000 parallel computer [ITS97] which was used to collect the results presented in Chapters 4 and 5. We also consider the AP3000's performance in terms of its communication latency and the degree to which it is possible to overlap communication and computation.

B.2 Node Architecture

The Fujitsu AP3000 parallel server is a distributed memory parallel computer based on a grid of 64-bit UltraSPARC nodes connected by a dedicated high-speed network (the AP-net). Fig. B.1 shows this architecture and the structure of each processing node. Each node runs the Solaris operating system and has a 300MHz UltraSPARC processor, 256MB RAM and a 4GB local disk with an uncached throughput of 6MB/s.

Access to the processing nodes is controlled by a queueing system. More specifically, the 60 processing nodes are partitioned into groups of up to 16 processors and user tasks are assigned to partitions by a control workstation. Client workstations use an external LAN connection to submit batch and interactive jobs to

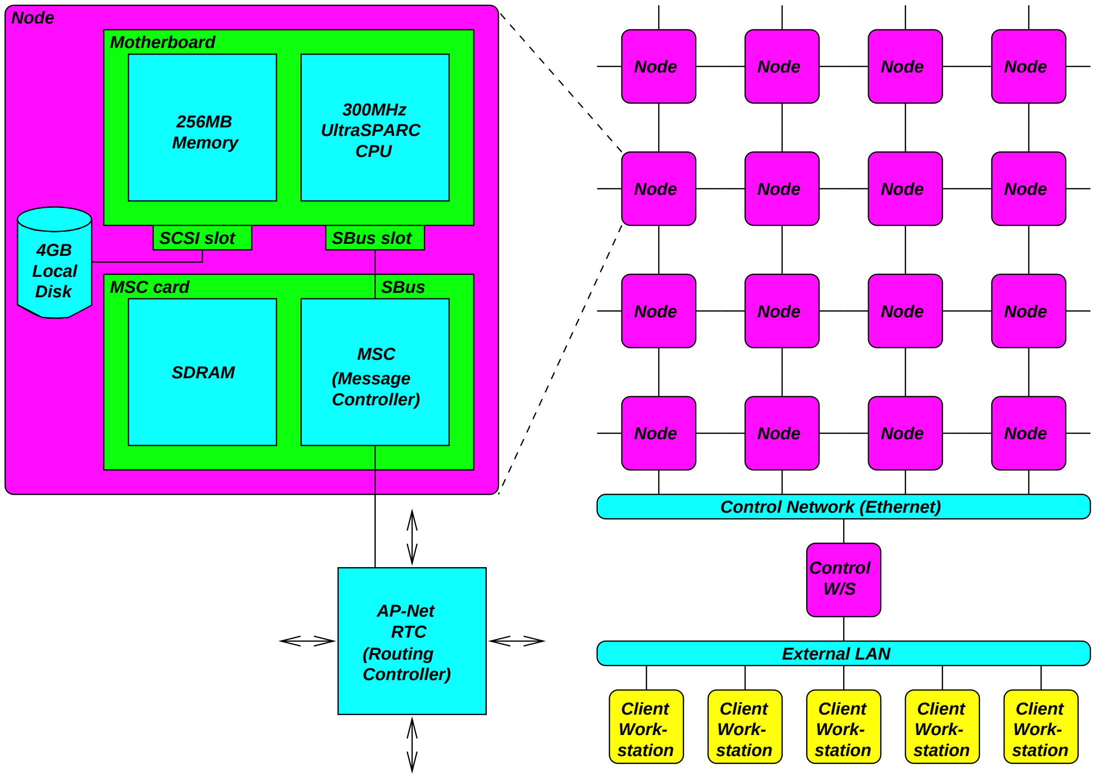


Figure B.1: AP3000 architecture.

the control workstation, which in turn makes use of a dedicated control network to map user tasks onto processing nodes. Batch jobs have sole use of all the processing nodes in the partition to which they were assigned.

B.3 Communication Network Architecture

The nodes of the AP3000 are connected into a 2D wraparound mesh topology by a high-speed network known as the AP-net. The AP-net is made up of Routing Controllers (RTCs) which are responsible for low-level message routing and barrier synchronisation operations. Each node is connected to an RTC by a Message Controller (MSC) card attached to a high speed I/O bus (the SBus).

To reduce communication latency, the RTCs support wormhole (or cut-through) routing. Under this scheme, messages are divided into small 4-byte pieces called flits. Flits in a message header determine the routing path of the message, and subsequent message flits are routed over the same path. The flits are pipelined, so intermediate processors do not wait for the entire message before forwarding incoming flits. Wormhole routing is faster than conventional store-and-forward routing and uses less memory.

User-level access to the AP-net is provided by a hierarchy of communication libraries, as shown in Fig. B.2.

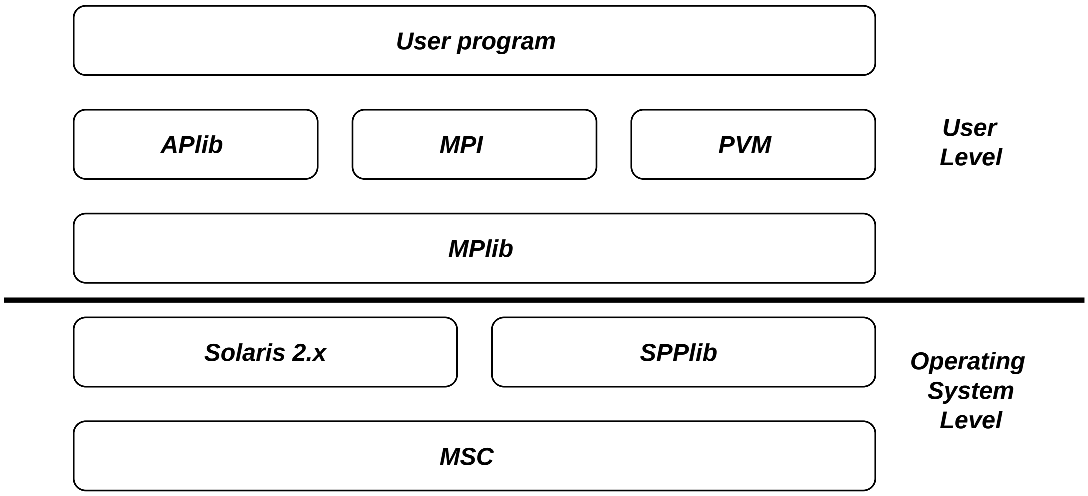


Figure B.2: AP3000 communications library hierarchy.

At the application level, users can chose from three different programming interfaces, viz. MPI (Message Passing Interface) [GLS94], PVM (Parallel Virtual Machine) [GBD+94] and APLib (Fujitsu's own communications library). These programming interfaces are based on a common hardware-independent library called MPLib, which implements reliable end-to-end message passing operations. In turn, MPLib makes use of SPPlib, a low-level communications library which provides direct access to the MSC hardware without the need for a system call. Services provided by SPPlib include MSC initialisation, unreliable send and receive (MPLib is responsible for higher-level services such as error checking), message buffer management and barrier synchronisation operations.

B.4 Performance Issues

B.4.1 Communication Cost

The time taken to communicate a message between two nodes in a 2D wraparound mesh with static wormhole routing is given by:

$$t_c(m) = t_s(m) + t_h(x, y, t_1, t_2) + t_b(m, \lambda)$$

where:

- $t_s(m)$ is the startup time required to prepare a message of length m bytes for transmission. This includes time taken to add headers, trailers, error correction information, as well as time to execute the routing algorithm. Our experiments on the AP3000 have shown that t_s depends on the message size m (in bytes) as follows:

$$t_s(m) = \begin{cases} (0.0116m + 88)\mu\text{s} & \text{if } m \leq 1024 \\ (0.0116m + 115)\mu\text{s} & \text{if } 1024 < m \leq 16384 \\ 310\mu\text{s} & \text{otherwise} \end{cases}$$

  The constant startup time for messages over 16K may be related to the size of the communications buffers used by the MSC card, many of which are 16K (or small integer multiples thereof). Longer messages may be repeatedly packed into these buffers in a pipelined fashion, thus avoiding the linear increase in latency experienced with smaller messages.

- $t_h(x, y, t_1, t_2)$ is the hop time required for a header to travel between the sender and the receiver. Here x and y are the dimensions of the mesh while t_1 and t_2 are the per-hop flit latencies for transfers in the same direction (i.e. X→X or Y→Y) and different directions (i.e. X→Y) respectively. For an $x \times y$ 2D wraparound mesh, the average number of hops between two randomly-selected (but distinct) processors $\bar{l}(x, y)$ is the sum of every possible communication distance divided by the number of possible destinations

$(xy-1)$, i.e.

$$\bar{l}(x,y) = \left(\sum_{i=-\lfloor \frac{x-1}{2} \rfloor}^{\lceil \frac{x-1}{2} \rceil} \sum_{j=-\lfloor \frac{y-1}{2} \rfloor}^{\lceil \frac{y-1}{2} \rceil} (|i| + |j|) \right) / (xy-1)$$

This simplifies to:

$$\bar{l}(x,y) = \frac{(x+y)(xy) - x(y \bmod 2) - y(x \bmod 2)}{4(xy-1)}$$

where $a \bmod b$ denotes the remainder when a is divided by b. Since the batch partition that we used to collect results allocates processing nodes at random from an 8×4 grid, we will use a value of $\bar{l}(8,4) \approx 3.10$.

The AP3000 uses a routing strategy called XY routing which routes messages in two phases – first in the X direction, then in the Y direction. Transfers in the same direction (X→X or Y→Y) require $t_1 = 120$ns while transfers which change direction (X→Y) require $t_2 = 170$ns. Exactly one change of routing direction is required whenever the sender and receiver are not in the same mesh column or row. If the sender and receiver are selected at random, this event occurs with probability:

$$p_h(x,y) = \frac{(x-1)(y-1)}{xy-1}$$

The average hop time (in seconds) is therefore given by:

$$t_h(x,y,t_1,t_2) = \bar{l}(x,y)t_1 + (t_2 - t_1)p_h(x,y)$$

For the particular case of the AP3000's 8×4 wraparound mesh, we have $\bar{l}(8,4) \approx 3.10, p_h(8,4) \approx 0.677, t_1 = 120$ns and $t_2 = 170$ns giving a value of $t_h \approx 405$ns.

- $t_b(m, \lambda)$ is the transmission time that a message of size m bytes requires to traverse the link between the sender and receiver, given that the link has an underlying throughput of λ bytes per second and uses cut-through routing. The AP-net's RTCs are theoretically capable of transferring data at up to 200MB/s. In practice, however, the performance of the SBus (which connects the MSC to the RTC) limits effective throughput to a measured maximum of $\lambda = 65.6$ MB/s. We therefore use a value of

$$t_b(m, \lambda) = \lambda m = (65.6 \text{ MB/s})^{-1} m \approx 14.89m \text{ ns}$$

As with many modern parallel computers, the hop time t_h of the AP3000 is very small compared to the setup time t_s and message transmission time t_b, even for small values of m. Ignoring the term gives:

$$t_c(m) = t_s(m) + t_b(m, \lambda)$$

From this simple model of the communication cost, the throughput $T(m)$ obtained when sending a message of m bytes between two processors on the AP3000 can be calculated as:

$$T(m) = \frac{m}{t_c(m)}$$

Fig. B.3 assesses the accuracy of this model by comparing observed and modelled throughputs for various messages sizes on the AP3000. Each of the observed results is a throughput average calculated from 25 measurements of the round trip time taken to transmit a message from one node to another and back again. The results predicted by the model agree with the observed throughputs to an accuracy of within 3% for messages longer than 512 bytes.

B.4.2 Overlapping Communication and Computation

An important requirement for efficient parallel programming is the ability to hide communication latency by overlapping communication and computation. This feature can yield dramatic performance improvements, especially when dealing with sparse, irregular problems that run on machines where communication is handled autonomously by an intelligent communication controller.

The MPI standard supports the overlap of communication and computation via the calls `MPI_Isend()` and `MPI_Irecv()` which initiate (but do not complete) send and receive operations. While these non-blocking operations are in progress,

m	$T(m)$ (MB/s)		%
	Obs.	Model	err.
64	0.843	0.697	-17.32
128	1.493	1.368	- 8.37
256	2.315	2.638	13.95
512	4.940	4.923	- 0.35
1K	8.574	8.686	1.31
1088	7.505	7.388	- 1.56
2K	12.113	11.817	- 2.44
4K	17.828	17.898	0.39
8K	24.271	24.098	- 0.71
16K	28.457	29.146	2.42
32K	40.176	40.110	- 0.16
64K	49.409	49.782	0.75
128K	56.711	56.607	- 0.18
256K	60.990	60.772	- 0.36
512K	62.664	63.094	0.69
1MB	64.183	64.323	0.22

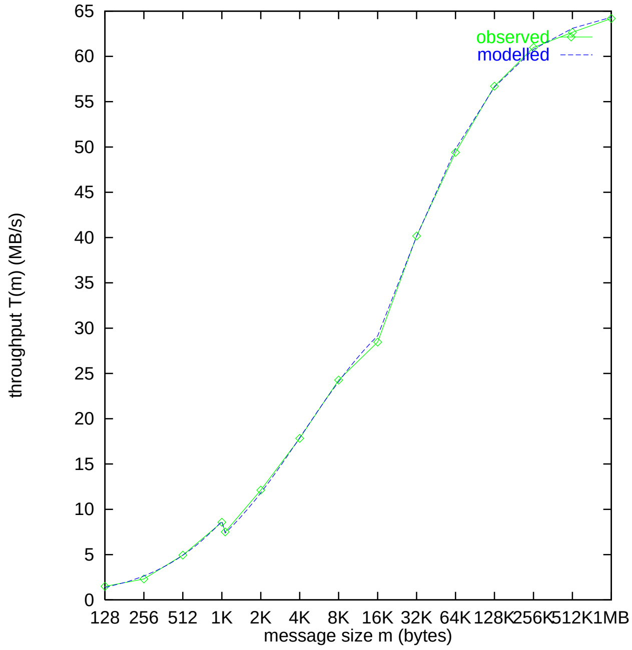


Figure B.3: Observed and modelled values of AP3000 throughput $T(m)$ (in MB/sec) in terms of the message size m.

programs proceed with other instructions. The completion of a non-blocking operation can be detected by using the `MPI_Test()` function; alternatively programs can wait for a non-blocking call to complete by using the `MPI_Wait()` function.

Experiments on the AP3000 reveal two major deficiencies in Fujitsu's MPI implementation and hardware architecture:

- While the non-blocking receive operation `MPI_Irecv()` behaves correctly and returns from the call almost immediately, the non-blocking send operation `MPI_Isend()` does in fact block. The blocking time is similar to the time taken for the blocking `MPI_Send()` call, which suggests that the two calls share the same implementation.

- Despite the presence of a dedicated Message Controller (MSC) card with DMA facilities, communication still requires a large amount of CPU intervention. CPU utilization during non-blocking operations is such that virtually no user instructions are executed while data is being actively sent or received.

Similar experiments conducted on an IBM SP-2 show that the SP-2 does not suffer from these problems. The SP-2's non-blocking send call returns almost immediately and background communication takes place with a relatively small amount of CPU overhead. Engineers from Fujitsu have acknowledged these problems with the AP3000 and say they are caused by the complex nature of MPI transfer protocols. At present they have no plans to address the problem since doing so would involve a complete redesign of the APnet device driver and MPI library.

An attempt to circumvent the blocking `MPI_Isend()` problem by implementing a true non-blocking send operation using light-weight threads was partially successful. However, the large amount of CPU intervention during the send operation severely limits opportunities for concurrency. In addition, Fujitsu's MPI library is not thread-safe, so correct operation under this system cannot be guaranteed.

As a result of these problems, the design of the steady-state solver described in Chapter 5 was oriented towards minimizing communication by exploiting the nearly lower-triangular structure of the transition matrix. Note, however, that the matrix vector kernel still allows for the overlapping of communication and computation on machines that have effective support for this feature.

Bibliography

[ACB84] 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.

[AH97] S.C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. In *Lecture Notes in Computer Science 1253*, pages 207–218. Springer Verlag, 1997.

[AK93] Haruo Akimaru and Konosuke Kawashima. *Teletraffic: theory and applications*. Springer Verlag, 1993.

[AKH97] S. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In *Proceedings of the 7th International Conference on Petri Nets and Performance Models (PNPM '97)*, pages 112–121. IEEE Computer Society Press, 1997.

[AMS90] S.F. Ashby, T.A. Manteuffel, and P.E. Saylor. A taxonomy for conjugate gradient methods. *SIAM Journal on Numerical Analysis*, 27(6):1542–1568, December 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar89] V. A. Barker. Numerical solution of sparse singular systems of equations arising from ergodic Markov chains. *Communications in Statistics: Stochastic Models*, 5(3):335–381, 1989.

[Bau93] 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 (PNPM '93)*. IEEE Computer Society Press, October 1993.

[BBC$^+$94] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.

[BCDK97] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of Kronecker operations on sparse matrices with applications to solution of Markov models. ICASE Report 97-66, NASA Langley Research Center, Hampton, VA, December 1997.

[BCM$^+$92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCMP75] 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.

[BDG94] M. Bernardo, L. Donatiello, and R. Gorrieri. Modelling and analyzing concurrent systems with MPA. In *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 89–106. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Regensberg/Erlangen, July 1994.

[BDMC+94] P. Buchholz, J. Dunkel, B. Müller-Clostermann, M. Sczittnick, and S. Zäske. *Quantitative Systemanalyse mit Markovschen Ketten.* Teubner, 1994.

[BFG+97] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[BFK99] P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99)*, pages 76–95, Zaragoza, Spain, September 1999.

[BK95] F. Bause and P.S. Kritzinger. *Stochastic Petri net theory.* Verlag Vieweg, Wiesbaden, Germany, 1995.

[Bou95] B. Boulter. Performance evaluation of HPF for scientific computing. In *Lecture Notes in Computer Science 919*. Springer Verlag, 1995.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS87] H. Beilner and F.J. Stewing. Concepts and techniques of the performance modelling tool HIT. In *Proc. European Simulation Multiconference*, Vienna, 1987.

[Buc94a] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems*, 15(1):59–80, 1994.

[Buc94b] P. Buchholz. Markovian Process Algebra: composition and equivalence. In *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling.* Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Regensberg/Erlangen, July 1994.

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

[Buc99] P. Buchholz. Projection methods for the analysis of Stochastic Automata Networks. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99)*, pages 149–168, Zaragoza, Spain, September 1999.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of OBBDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.

[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.

[CM99] G. Ciardo and A.S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proceedings of the 8th International Conference on Petri Nets and Performance Models (PNPM '99)*, pages 22–31, Zaragoza, Spain, September 1999. IEEE Computer Society Press.

[CMT91] G. Ciardo, J.K. Muppula, and K.S. Trivedi. On the solution of GSPN reward models. *Performance Evaluation*, 12(4):237–253, 1991.

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

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DCB93] T. Demaria, G. Chiola, and G. Bruno. Introducing a color formalism into Generalised Stochastic Petri nets. In *Proceedings of the 9th International Workshop on Application and Theory of Petri Nets*. IEEE, October 1993.

[DFG83] 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.

[Don94] S. Donatelli. Superposed stochastic automata: a class of stochastic Petri nets amenable to parallel solution. *Performance Evaluation*, 18:21–36, 1994.

[DS97] D.D. Deavours and W.H. Sanders. An efficient disk-based tool for solving very large Markov models. In *Lecture Notes in Computer Science 1245: Proceedings of the 9th International Conference on Modelling, Techniques and Tools (TOOLS '97)*, pages 58–71, St. Malo, France, 3–6 June 1997. Springer Verlag.

[DS98a] D.D. Deavours and W.H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33(1):67–84, June 1998.

[DS98b] D.D. Deavours and W.H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.

[Fer98] P.H.L. Fernandes. *Méthodes Numériques pour La Solution de Sysèmes Markoviens à Grand Espace d'États*. PhD thesis, Institut National Polytechnique de Grenoble, February 1998.

[FGN92] R.W. Freund, G.H. Golub, and N.M. Nachtigal. Iterative solution of linear systems. *Acta Numerica*, pages 1–44, 1992.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In *Lecture Notes in Mathematics*, volume 506, pages 73–89. Springer Verlag, 1976.

[FM84] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, April 1984.

[FMY97] M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representations. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[FN91] R.W. Freund and N.M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[FPS98] P. Fernandes, B. Plateau, and W.J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[Fre93] R.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.

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

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Massachussetts, 1994.

[Gia99] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College, London, January 1999.

[GKS95] A. Gupta, V. Kumar, and A. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):455–469, May 1995.

[GL87] L. Goldschlager and A. Lister. *Computer Science: A Modern Introduction.* Prentice Hall International Series in Computer Science. Prentice Hall, 1987.

[GL89] G.H. Golub and C.F. van Loan. *Matrix Computations.* John Hopkins Press, Maryland, 2nd edition, 1989.

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

[God96] P. Godefroid. *Lecture Notes in Computer Science 1032: Partial-Order Methods for the Verification of Concurrent Systems: An approach to the state explosion problem.* Springer Verlag, 1996.

[Goo88] R. Goodman. *Introduction to Stochastic Models.* Benjamin/Cummings, 1988.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Lecture Notes in Computer Science 697: Proceedings of Computer Aided Verification '93 (CAV '93)*, pages 438–449. Springer Verlag, 1993.

[Gut93a] M.H. Gutknecht. Changing the norm in conjugate gradient type algorithms. *SIAM Journal on Numerical Analysis*, 30(1):40–56, February 1993.

[Gut93b] M.H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM Journal on Scientific Computing*, 14(5):1020–1033, September 1993.

[Her99] H. Hermanns. *Interactive Markov Chains.* PhD thesis, Universität Erlangen-Nürnberg, September 1999.

[Hil94] J. Hillston. *A Compositional Approach to Performance Modelling.* PhD thesis, University of Edinburgh, 1994.

[HL94] G. Horton and S. Leutenegger. A multi-level solution algorithm for steady-state Markov chains. In *Proceedings of the SIGMETRICS '94*, Nashville, Tennessee, 1994.

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

[HMPS96] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1479–1493, December 1996.

[HMS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99)*, pages 188–207, Zaragoza, Spain, September 1999.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoc96] Ng Chee Hock. *Queueing Modelling Fundamentals.* John Wiley and Sons, 1996.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[Hol95] 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.

[HP93] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* International Computer Science Series. Addison Wesley, 1993.

[HR94] H. Hermanns and M. Rettelbach. Syntax, semantics, equivalences and axioms for MTIPP. In *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling.* Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Regensberg/Erlangen, July 1994.

[HR98] J. Hillston and M. Ribaudo. Stochastic process algebras: a new approach to performance modelling. In K. Bagchi and G. Zobrist, editors, *Modelling and Simulation of Advanced Computer Systems*, chapter 10, pages 235–256. Gordon Breach, 1998.

[HS52] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–435, 1952.

[HS99] P.G. Harrison and B. Strulo. Spades: a process algebra for discrete event simulation. *Journal of Logic and Computation*, 1999.

[HY81] L. A. Hageman and D. M. Young. *Applied Iterative Methods.* Academic Press, 1981.

[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.

[Kel79] F.P. Kelly. *Reversibility and Stochastic Networks.* John Wiley and Sons, 1979.

[Kem96] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(9):615–628, 1996.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings Publishing, 1994.

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

[KHMK99] W.J. Knottenbelt, P.G. Harrison, M.A. Mestern, and P.S. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation*, 1999. To appear.

[KK98] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998. URL `http://www.cs.unm.edu/~karypis`.

[Kle75] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley and Sons, 1975.

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

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

[Knu98] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

[KPV97] I. Kokkarinen, D. Peled, and A. Valmari. Relaxed visibility enhances partial order reduction. In *Lecture Notes in Computer Science 1254: Proceedings of Computer Aided Verfication '97 (CAV '97)*, pages 328–339. Springer Verlag, 1997.

[KS60] J.G. Kemeny and J.L. Snell. *Finite Markov Chains.* Van Nostrand, 1960.

[Kul95] V.G. Kulkarni. *Modeling and Analysis of Stochastic Systems.* Chapman & Hall, 1995.

[LG93] J.G. Lewis and R.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.

[Lin98] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets.* John Wiley and Sons, 1998.

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

[MCC97] P. Marenzoni, S. Caselli, and G. Conte. Analysis of large GSPN models: a distributed solution tool. In *Proceedings of the 7th International Conference on Petri Nets and Performance Models '97 (PNPM 97)*, pages 122–131. IEEE Computer Society Press, 1997.

[Mes98] Mark A. Mestern. Distributed analysis of Markov chains. Master's thesis, University of Cape Town, Cape Town, South Africa, April 1998.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPS96] V. Migallón, J. Penadés, and D.B. Szyld. Block two-stage methods for singular systems and Markov chains. *Numerical Linear Algebra with Applications*, 3:413–426, 1996.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

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

[Pet81] J.L. Peterson. *Petri Nets and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Pla85] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. *Performance Evaluation Review*, 13:142–154, 1985.

[PRCB94] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using Boolean manipulation. In *Lecture Notes in Computer Science 815*, pages 416–435. Springer Verlag, 1994.

[Rab80] M.O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley and Sons, 1988.

[Rei92] W. Reisig. *A Primer in Petri Net Design.* Springer Verlag, 1992.

[RS94] M. Rettelbach and M. Siegle. Compositional minimal semantics for the stochastic process algebra TIPP. In *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 31–50. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Regensberg/Erlangen, July 1994.

[RSBS96] R.K. Ranjan, J.V. Sanghavi, R.K. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *Proc. International Conference on Computer Design (ICCD '96)*, pages 358–364, 1996.

[Saa89] Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.

[Sal98] D. Salomon. *Data Compression: The Complete Reference*. Springer Verlag, 1998.

[Scz87] M. Sczittnick. Technicken zur funktionalen und quantitativen Analyse von Markoffschen Rechensystemmodellen. Diplomarbeit, Universität Dortmund, October 1987.

[SD95] 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.

[SD97] U. Stern and D.L. Dill. Parallelizing the Murφ verifier. In *Lecture Notes in Computer Science 1254: Proceedings of Computer Aided Verification '97 (CAV '97)*, pages 256–267. Springer Verlag, 1997.

[SD98] U. Stern and D.L. Dill. Using magnetic disk instead of main memory in the Murφ verifier. In *Lecture Notes in Computer Science 1427: Proceedings of Computer Aided Verification '98 (CAV '98)*, pages 172–183. Springer Verlag, 1998.

[SF93] G.L. Sleijpen and D.R. Fokkema. BiCGSTAB(L) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, 1:11–32, September 1993.

[She93] G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.

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

[SS86] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

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

[Ste97] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration*. PhD thesis, Technischen Universität München, October 1997.

[Sto95] A.L. Stornetta. Implementation of an efficient parallel BDD package. Master's thesis, University of California Santa Barbara, 1995.

[Str93] B. Strulo. *Process Algebra for Discrete Event Simulation*. PhD thesis, Imperial College, London, October 1993.

[SV86] A. van der Sluis and H.A. van der Vorst. The rate of convergence of conjugate gradients. *Numerische Mathematik*, 48:543–560, 1986.

[SV95] G.L. Sleijpen and H. van der Vorst. An overview of approaches for the stable computation of hybrid Bi-CG methods. *Applied Numerical Mathematics*, 19:235–254, 1995.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990, Lecture Notes in Computer Science 483*, pages 491–515. Springer Verlag, 1991.

[Val98] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1491*, pages 429–528. Springer Verlag, 1998.

[Var62] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

[Vor92] H. 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.

[Wei94] R. Weiss. Orthogonalization methods. Interner Bericht 52, Universität Karlsruhe, Rechenzentrum der Universität Karlsruhe, April 1994.

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

[WL91] C.M. Woodside and Y. Li. Performance Petri net analysis of communication protocol software by delay-equivalent aggregation. In *Proceedings of the 4th International Workshop on Petri nets and Performance Models*, pages 64–73, Melbourne, Australia, 2–5 December 1991. IEEE Computer Society Press.

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Lecture Notes in Computer Science 697*, pages 59–70. Springer Verlag, 1993.

[Yan94] U. Meier Yang. Preconditioned conjugate gradient-like methods for nonsymmetric linear systems. Technical Report 1210, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1994.